UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

Agile Testing. Estado del arte. Su aplicación en empresas TIC de Extremadura.

Autor: Antonio Félix Sánchez-Oro Portillo

Tutor: Amparo Navasa Martínez

Fecha: Julio de 2017

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

Agile Testing. Estado del arte. Su aplicación en empresas TIC de Extremadura.

Autor: Antonio Félix Sánchez-Oro Portillo

Tutor: Amparo Navasa Martínez

TRIBUNAL:

Presidente: Miguel Ángel Pérez Toledano

Secretario: María Ángeles Mariscal Araujo

Vocal: Miryam Salas Sánchez

"Quiero dedicar este TFG a mi familia,

por darme todo vuestro apoyo, esfuerzo, y paciencia.

Por confiar en mí.

Vosotros habéis cruzado la meta conmigo.

A ti, Cristina, que me has dado tanto en tan poco,

que eres mi compañera de viaje, que me ayudas a seguir y perseverar.

Y a mi tutora Amparo, por hacer posible este TFG."

Contenido

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

RESUMEN

Con la aparición de las metodologías ágiles, ha surgido una nueva tendencia de realizar pruebas de software. En este nuevo tipo de pruebas, los test son los protagonistas del desarrollo del software y el código pasa a un segundo plano, con la promesar de ser más eficaces en la detección de errores.

Este Trabajo Fin de Grado (TFG) trata de explicar que se entiende por pruebas ágiles y cómo se aplican. Por otra parte, hemos hecho un estudio para comprobar su aplicación en Extremadura. Para llegar a este punto, se explicarán los diferentes tipos de test que existen, las metodologías ágiles más utilizadas y finalmente, se acabará por estudiar los distintos tipos de pruebas ágiles más usados hoy en día.

ABSTRACT

With the advent of agile methodologies, a new tendency to perform software testing has emerged. In this new type of tests, they are the protagonists of software development and the code goes to background, with the promise to be more effective in detecting errors.

This End of Grade Work (TFG) tries to explain what is meant by agile tests and how they are applied. On the other hand, we have done a study to verify its application in Extremadura. To get to this point, we will explain the different types of tests that exist, the most used agile methodologies and finally, will end up studying the different types of agile tests used today.

1. INTRODUCCIÓN

Cuando empecé mis estudios de Ingeniería Informática, el mundo que rodea a los aspectos de calidad del software era inexistente y desconocido para mí. Poco a poco fui descubriendo el enorme y complejo ecosistema, dentro de la ingeniería informática, que rodea la calidad de los proyectos de desarrollo software y es eso justamente, lo que me hizo escoger este Trabajo Fin de Grado (TFG).

Me parece uno de los aspectos más importantes, o el que más, a tener en cuenta a día de hoy en el desarrollo de proyectos software, y sin embargo parece que en muchos programas se menosprecia la calidad. Cada vez salen al mercado más productos sin probar debidamente, y se percibe una falta de calidad que se está convirtiendo en habitual. ¿Por qué ha cambiado esto?

Este TFG no trata de la historia de la informática, ni de sus inicios, ni cómo ha ido evolucionando el software, sin embargo, sí quiero poner un ejemplo que sirva para entender la deriva de la situación actual en este ámbito:

Dentro del abanico de posibilidades que podemos encontrar en los productos de software voy a centrarme, para este ejemplo, en uno muy concreto: la industria de los videojuegos.

En 1994 este sector informático estaba formado por proyectos pequeños fácilmente controlables y con pocas variables a tener en cuenta. Los productos de este tipo salían al mercado para diferentes plataformas sin fallos o con una tasa de fallos tan pequeña que el cliente difícilmente podía apreciarlos. Uno de los mayores motivos de hacer un producto “perfecto” era que no se podía parchear después puesto que, por ejemplo, en plataformas cerradas (consolas) no era posible tener conexión a internet.

Sobre el año 2000 la industria del videojuego se había embarcado en proyectos no tan pequeños, bastante más complejos y con un gran número de variables y situaciones que no eran tan fáciles de resolver técnicamente, como pasaba antes.

Aun así, la calidad era buena y en el peor de los casos, suficiente. Aún existía la exigencia de crear productos libres de defectos puesto que no había manera de actualizarlos.

En la actualidad, hay muchos títulos en los que encontramos una gran ausencia de calidad. Con la llegada de internet, parece que, una gran parte de los desarrolladores

ha olvidado cómo hacer buenos proyectos sin fallos. Ahora una gran parte de los productos software, se lanzan al mercado, incompletos o no suficientemente probados[1], de modo que es necesario (vía internet) parchearlos y actualizarlos[2]. Hay títulos que tienen fallos graves que nunca han llegado a solucionarse[3], por no hablar de los proyectos en marcha que han tenido que ser cancelados porque los creadores no previeron una dificultad tan grande al crearlos (1)[4].

Así, nos encontramos con dos tipos de problemas graves. El primero surge al inicio del proyecto: no presupuestar bien, no delimitar el alcance del proyecto o no tener claro los objetivos a cumplir, hacen que cada vez haya cancelaciones más sonadas. El segundo son los problemas que aparecen una vez finalizado el desarrollo del proyecto: cada vez se ven más casos de productos finalizados que están incompletos o tienen partes deficientes en cuando a calidad se refiere. Esto se hace patente en la cantidad de actualizaciones que tiene ese producto o directamente, en fallos apreciables durante la reproducción del juego. Los proyectos no están terminados. ¿Nos podríamos imaginar ir al taller cada mes porque el coche tiene fallos que hay que ir solucionando sobre la marcha y no se habían detectado antes por falta de pruebas? No volveríamos a comprar un coche a ese mismo fabricante. Es, simplemente, inaceptable.

Los productos de este tipo, los videojuegos, son cada vez más complejos y ahora con internet es fácil parchearlos. Esto no es una solución elegante, es práctica. Por eso se ha bajado la guardia en aspectos de calidad.

No es objeto de este TFG analizar con detalle los diferentes ámbitos relativos a la calidad de software, ni revisar todas las metodologías existentes. Pero para llegar al objetivo final, parece conveniente recorrer de manera general, los pasos que fueron

---

[1] [1] F1 2010 de Codemasters*: "El juego ya es conocido en los foros y comunidades de jugadores en Internet por sus numerosos fallos que impiden el buen funcionamiento de éste y que ha decepcionado a una gran parte de usuarios. Según la opinión de los mismos, no ha sido testado, y se trata de una versión pre-alpha. Entre los errores destacan problemas en el guardado de la partida, en los pitstops y errores en un multijugador no muy desarrollado. Cuenta además con numerosos errores gráficos, como los retrovisores de algunos monoplazas"* (108) (109)

[2] Mafia III: Recibió su primer parche cinco días después de ser lanzado por el famoso "framerate" (El juego no era capaza de correr a más de 30fps). (110)

[3] Empire Total War: SEGA no llegó a solucionar fallos críticos del juego, haciendo imposible completar las partidas. El juego terminó su soporte técnico con el fallo sin solucionar.

[4] Silent Hill PS4 de Konami: Aparte de los problemas económicos, también tuvieron problemas técnicos. La desarrolladora no pudo abarcar la gran complejidad del juego con el presupuesto inicial.

dando las metodologías tradicionales antes de llegar a los desarrollos ágiles. Así llegaremos a explicar la necesidad de su existencia para después, centrarnos en un aspecto concreto, las metodologías ágiles y dentro de éste, comprobar cómo están evolucionando los mecanismos de pruebas, para afrontar proyectos que cada vez son más complejos dentro de este paradigma.

1.1. OBJETIVOS

Este trabajo tiene dos bloques diferenciados con dos objetivos distintos. Por un lado, se encuentra la **parte teórica** donde: se realiza un repaso histórico, se explica qué es el ciclo de vida y en qué consiste el control de calidad del software. En esta parte también se presentan las pruebas de software más utilizadas hoy en día (CAPÍTULO 3). Después se pasa a explicar la creación y el funcionamiento de las metodologías ágiles, y se estudia en mayor profundidad dos de ellas, SCRUM y XP (CAPÍTULO 4).

Y para finalizar la parte teórica, se estudia las técnicas existentes de *agile testing* (CAPÍTULO 5). El objetivo de este capítulo, es estudiar y presentar las herramientas y técnicas utilizadas para ejecutar pruebas ágiles. Para llegar a este objetivo es necesario presentar con antelación, que es y cómo surgen los métodos ágiles. Además, no es correcto explicar las técnicas ágiles si no se apoyan sobre una metodología. Por esta razón en trabajo se describe el funcionamiento de dos metodologías agiles en el capítulo 4. Por un lado, se explica el funcionamiento de Scrum, por ser el método más usado y, por otro lado, se explica Extreme Programming, por ser el precursor de las pruebas ágiles.

En la **parte práctica**, existe un estudio realizado a partir de datos de empresas de Extremadura (CAPÍTULO 6). El trabajo de campo, el cual se resume en el segundo bloque, tiene otro objetivo diferente. En esta parte se contacta con empresas para obtener información sobre cómo trabajan y qué métodos utilizan para desarrollar software. El objetivo de este estudio es comprobar de qué manera desarrollan software las empresas de nuestro entorno y, en particular, si realizan pruebas ágiles y o no. El último capítulo de este TFG es una serie de conclusiones extraídas de los datos cualitativos del estudio y una breve opinión personal con la única motivación de proponer un debate sobre el estado del sector TIC en Extremadura.

Finalmente hay una serie de anexos, dónde se puede encontrar:

a) un "*paper*" hecho para la Universidad de La Laguna, donde exponemos de manera inicial en qué consiste el estudio y cómo lo vamos a realizar. Este artículo finalmente no fue escogido por estar el estudio poco avanzado (hay que tener en cuenta que se envió en febrero del 2017 y en esa fecha el estudio estaba comenzando y no se disponía de datos concluyentes).

b) El modelo de entrevista personal, usado para obtener los datos del estudio.

c) Un modelo de consentimiento firmado donde se le informa al entrevistado que dispone de derechos ARCO para la información publicada en este TFG.

d) El formulario usado tanto en las entrevistas personales, como en el cuestionario online.

2. DESARROLLO SOFTWARE

En este capítulo se va a comentar el surgimiento de la ingeniería de software, qué son los ciclos de vida del software, cómo funciona el control de calidad y estudiar los distintos tipos de pruebas o test que existen para detectar los errores en el desarrollo de un producto software.

2.1. REPASO HISTÓRICO

El inicio del desarrollo de software, aunque parezca muy lejano, sólo se encuentra a pocas décadas de distancia. En 1965, cuando la programación usaba técnicas de codificación y desarrollo primitivas, comenzó un periodo caótico denominado Crisis del Software, que terminaría en 1985. Pero fue en 1968 durante una conferencia de la OTAN (2)[5], donde se discutió por primera vez acerca de la productividad y calidad del software. En esa época no había métodos, ni técnicas definidas sobre cómo llevar a cabo un proyecto software con éxito. La mayoría de los desarrollos se implementaban sin una planificación previa y con lenguajes de programación primitivos (recordemos que la programación orientada a objetos fue muy posterior, a mediados de los 80). Como consecuencia de esta situación, la mayoría de los desarrollos fracasaban, ya que a veces no llegaban ni a ser funcionales ni satisfacían al cliente.

Al no tener un plan de desarrollo, la implementación se hacía "en espagueti". En proyectos largos, llegaba un momento en el que había tanto código que a los desarrolladores se les hacía imposible continuar su tarea: el problema era demasiado complejo. En esta reunión de la OTAN, se debatió acerca de esta situación y analizaron a qué tipo de problema se enfrentaban: ¿era un problema de índole matemático? ¿semántico? ¿de preparación técnica? La respuesta fue simple y concisa: el tipo de problema con el que estaban lidiando, era un problema de ingeniería. A partir de aquí, se copiaron prácticas de otras ingenierías y se aplicaron al desarrollo de software.

Esta cumbre de 1968, y la posterior de 1969, resultó ser un punto de inflexión, pues por primera vez surge el término de ingeniería del software de manera formal, y provocó que poco después surgiera la metodología en cascada (o clásica) que definiría por primera vez una serie de reglas y técnicas para desarrollar software, minimizando

[5] Crisis del Software: Fue aquí donde se creó la rama de Ingeniería Informática (1968).

los errores y aumentado así la calidad del producto final. En torno a 1990 este modelo quedó obsoleto, ya no era suficientemente efectivo para la complejidad de los proyectos de entonces.

En 1994 Standish Group denunció esta situación, señalando como culpable a la metodología en cascada, y criticando duramente la eficacia de este ciclo de vida (3). En este informe, se indicaba que solamente el 16% de los desarrollos software se completaban con éxito, el 31% de los proyectos se cancelaban antes de empezar o que el 52% de los proyectos no cumplían el presupuesto inicial y se pasaban del tiempo estimado etc. Como consecuencia de esta situación tan perjudicial para la industria informática, surgen otros tipos de ciclos de vida, como el incremental o el iterativo, que consiguen aumentar la eficacia de los ciclos de vida y mejoraban el desarrollo, pero para muchas situaciones o productos concretos, seguía habiendo problemas graves.

Estos ciclos de vida son la antesala de las metodologías ágiles, que surgen a principios del año 2000, con el "manifiesto ágil". Esto se trata más detalladamente en el capítulo 4, de las metodologías agiles.

Este capítulo está enfocado al estudio de la situación actual en lo que se refiere a calidad del software. El software es un término abstracto, no podemos tocar los materiales de los que está formado y no se puede apreciar la calidad al tacto como en cualquier otro producto. Por lo tanto, cuando se habla de calidad del software, se hace referencia a un conjunto de características que son intangibles pero que cumplen una serie de requisitos, que pueden ser implícitos o no. Desde un punto de vista menos formal, podemos afirmar que la calidad del software realmente, es el nivel de satisfacción que tiene un usuario cuando un producto software satisface sus necesidades (4).

Para alcanzar tal objetivo, es necesario llevar a cabo un proceso de supervisión antes de lanzar un producto al mercado. Esta supervisión es, fundamentalmente, realizar pruebas que garanticen que el producto software cumpla con todas sus funcionalidades al completo y sin problemas.

2.2. EL CICLO DE VIDA

Desde un punto de vista clásico, la mayoría de los procesos de desarrollo software tienen las mismas etapas, aunque cambien las formas de ejecución. Esto es el estándar ISO/IEC 12207[6]. En otras palabras, el ciclo de vida de un software, de manera tradicional, tiene unas etapas determinadas y una ejecución lineal, de manera que se procede de la siguiente forma: una primera fase de análisis y planificación, otra de diseño y otra de implementación, seguida de una fase de pruebas y documentación para acabar finalmente con el lanzamiento del producto y su mantenimiento.

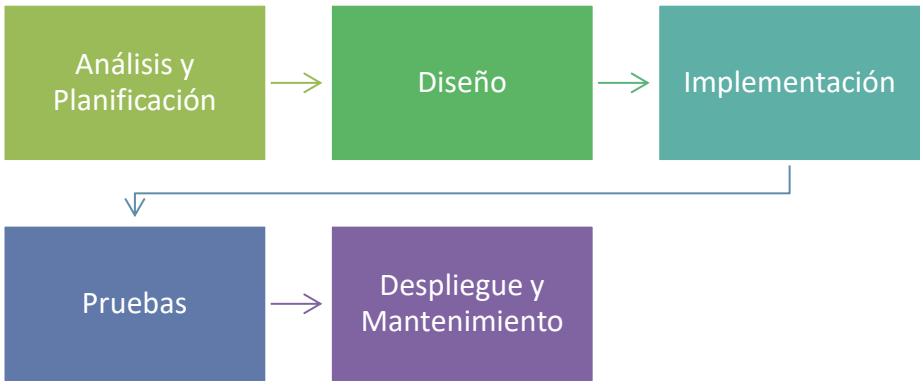

Ilustración 1: Ciclo de Vida Tradicional

A lo largo del tiempo han surgido muchos ciclos de vida, cada uno orientado a un enfoque diferente y creado para solucionar problemas derivados de cada tipo producto. No todos los proyectos son igual de complejos, de manera que su desarrollo no es el mismo. Cada producto a diseñar es diferente, y para cada uno de ellos conviene seguir un ciclo de vida adaptado a las necesidades del producto, con una forma de desarrollar software diferente en las formas y en el tiempo.

A pesar del abanico tan grande de posibilidades a escoger en los ciclos de vida, en este capítulo se comentará la metodología clásica que tradicionalmente se ha estado usando para desarrollar software.

---

[6] ISO/IEC 12207 Information Technology / Software Life Cycle Processes: *es el estándar para los procesos de ciclo de vida del software de la organización ISO.* (111)

Cuando hay un problema que se quiere resolver desde un punto de vista ingenieril, el primer paso es analizar la situación: tanto como si se quiere construir un edificio, como si se quiere diseñar un producto software, la primera fase es el análisis del problema. En el caso de la ingeniería de software, hay que llevar a cabo labores de análisis de los requisitos que tanto el sistema como el cliente, necesitan y demandan. Hay que analizar y tener claro qué lenguaje o lenguajes de programación son los más adecuados, qué entornos usar, preparar todas las herramientas necesarias, planificar tiempos de desarrollo, costes y contabilizar el capital humano y material que se necesita para abordar el todo el desarrollo.

Además del análisis, para la planificación del desarrollo, hay que analizar los diseños realizados del sistema que se va a implementar. Esto se realiza mediante la definición del flujo de datos, las estructuras de datos primarias, características de uso, objetivos a cumplir y restricciones del cliente. Esto se traduce en la creación artefactos tales como los casos de uso y los requisitos funcionales.

Acabada esta fase comienza una segunda etapa de diseño, en la cual se realizan labores de planificación de todo el "esqueleto" o arquitectura del sistema. En esta etapa se define el diseño de los datos, aquellas estructuras para soportar y almacenar los datos que usará el software en desarrollo, la arquitectura del software donde están definidos las entidades que la construyen y los procesos entre ellas, y finalmente la interfaz del usuario, que hará de conexión entre el software y el usuario.

La siguiente fase es implementar la solución que se ha diseñado. El equipo de desarrollo hará labores de codificación e implementación. Se creará el programa por primera vez, y deberá ejecutarse e interactuar con el usuario final. Esta fase está marcada por el número de programadores, siendo posible y recomendable la división en grupos y la jerarquización de éstos, como medida para aumentar la organización y la eficacia, si fuera necesario. Se debe realizar también la documentación del código, tanto interna (alojada en el código fuente) como externa (documentación que acompaña al software).

A continuación, se desarrolla la tarea que más tiempo debería llevar al equipo de desarrollo, las pruebas.

Es en esta etapa donde se prueba, de distintas formas, todo lo codificado. Más adelante veremos los tipos de pruebas que hay. En un principio lo normal es lanzar pruebas

unitarias y de integración. Se corrigen los fallos surgidos de las fases anteriores y se termina de documentar el código.

Una vez hayamos realizado el proceso de pruebas, se lanza el producto y se llevan labores de explotación y mantenimiento. También se corrigen los errores que pudieran quedar de las fases anteriores que no se descubrieron, y se implementa, si fuera necesario, nuevas funcionalidades. Si el entorno cambiase, por ejemplo: actualizaciones del sistema operativo, librerías, entorno web… se deberán implementar parches para corregir este deterioro de la aplicación.

A este ciclo de vida, que acabamos de explicar, se le llama en cascada ("*waterfall*"), y es el principal enemigo de las metodologías ágiles. Se considera obsoleto, y la tasa de fracaso al usar este método es alto. ¿Por qué?

Alguno de los problemas graves del desarrollo en cascada, reside en su poca capacidad de respuesta a cambios. Es un método férreo: si durante la fase de implementación, por ejemplo, hay algún pequeño cambio en los requisitos de uso, no habría margen de respuesta, puesto que este desarrollo no contempla la vuelta atrás, no sería posible reconsiderar las acciones tomadas en fases previas.

Que no podamos reconsiderar acciones ya tomadas en fases anteriores es un problema, pero también es otro problema grave el no poder repetir procesos de una fase: no se contempla la iteración de fases. ¿Y si hay errores en alguna de ellas? Esta situación obligaría a arrastrar los fallos a las siguientes etapas, y cuando se intenta depurarlos, ya pueden no ser subsanables. En este tipo de ciclo de vida, las pruebas se realizan al final del proceso de elaboración del software. Hay que tener en cuenta, que las pruebas son otra fase más y que posteriormente se realiza el lanzamiento a producción. No se contempla repetir la fase de pruebas, ni reconsiderar nada. Si se realiza una mala fase de pruebas lanzaremos el producto con errores, sí o sí.

Gracias a informes como el CHAOS REPORT (5), podemos comprobar que actualmente, este tipo de ciclo de vida en cascada no se aplica en la gran mayoría de desarrollos de software. Hay métodos de desarrollo surgidos posteriormente a éste que subsanan algunos de sus defectos. Uno de ellos es el modelo iterativo.

No se va a detallar aquí los modelos o ciclos de vida posteriores pues se sale del objetivo de este TFG. En este apartado sólo se ha pretendido mostrar a grandes rasgos el ciclo de vida clásico para entender y comparar, el avance que ha supuesto las

metodologías ágiles en lo que a calidad de software se refiere (Se retomará este ciclo de vida clásico en el CAPÍTULO 4).

2.3. EL CONTROL DE CALIDAD DEL SOFTWARE

Al comienzo del capítulo se mencionó una definición de calidad de software, pero al estudiar en profundidad este término surgen múltiples definiciones, que son complementarias entre sí y todas correctas. Con este argumento se deduce que "calidad en el software" es un término relativo y siempre va a depender de los requisitos del cliente.

Al final, el objetivo es el mismo: que los requisitos que demanda el cliente, los específicos del programa (aquellos que intentamos conseguir) y los que han sido alcanzados, coincidan de forma que el cliente, y en su defecto el usuario final, perciba esa "calidad intangible" que se propuso desde el comienzo del desarrollo.

Para lograr este objetivo, hay múltiples modelos teóricos propuestos, como el de Boehm o el de SQM (Software Quality Management) que se basan en tres partes: en primer lugar, se encuentran las métricas del producto, que son medidas cuantitativas sobre algunos atributos del software. Después, se abordan los criterios de calidad del producto, que son atributos que contribuyen de una u otra forma a lograr la calidad deseada. Por último, encontramos los factores de calidad del software, que representan el punto de vista del usuario final, en cuanto a calidad se refiere (6).

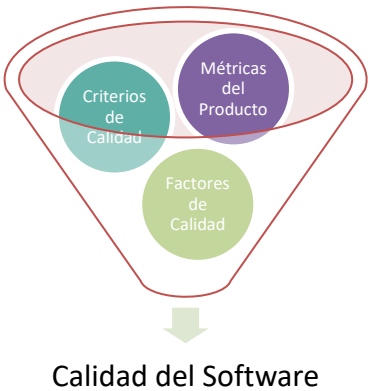

Ilustración 2: Estructura de un modelo de calidad genérico

Cualquier modelo de calidad no está vertebrado si no contiene técnicas que lo complementen. Las últimas tendencias combinan tareas de calidad con técnicas formales teniendo un impacto en la calidad del producto final, puesto que aumenta la rigurosidad, consistencia y completitud del desarrollo del producto, y evita defectos que estrictamente se originan en la programación del sistema (7).

Se puede hablar de tres tipos de técnicas: la primera consiste en usar técnicas matemáticas para comprobar que el software se ajusta a los parámetros correctos. A esto se le denomina **verificación formal**.

La segunda se conoce como **garantía de calidad estadística.** Utiliza datos estadísticos para encontrar causas que originen errores o defectos en el software: se recopila información de los defectos en el software y se establece una clasificación, después se determina el origen de los fallos y se selecciona la causa que originó más fallos. Con estos datos, se prevé y se comprueba que el software en desarrollo no ha producido los mismos fallos que otros procesos de desarrollo similares.

La última técnica es una combinación de las dos anteriores y se llama **proceso limpio**. El objetivo de esta última técnica es prever posibles errores que pudieran surgir, para así minimizar el trabajo de depuración posterior. Esta técnica proporciona una medida adicional de calidad: un índice de defectos.

El cumplimiento de este conjunto de características deseables lleva a alcanzar o mejorar la calidad de software. En los siguientes apartados se muestran los tipos y características de las pruebas del software.

2.4. PRUEBAS DE SOFTWARE

Para probar completamente un producto software antes de su lanzamiento al mercado, lo ideal sería probar todas y cada una de las líneas de código, o dicho de otro modo, se debería haber probado todos los caminos posibles que pudiera llegar a tomar el programa en ejecución, considerando todos los valores posibles. En teoría es el plan perfecto a seguir si queremos probar completamente un software, pero a la hora de llevarlo a cabo nos topamos con la realidad. El plan perfecto es irrealizable. Esto se comprende con mayor claridad con un ejemplo: considérese una función que realice la suma de dos enteros. Para probarla completamente, habría que insertar todos los posibles valores de entrada a la función y comprobar si el valor de salida concuerda

con el valor esperado. Esto no se puede realizar así. Solamente con pensar en el tiempo que se tardaría en probar todas las posibles combinaciones, podemos ver, que probar exhaustivamente un producto software es, cuanto menos impensable, aunque se trate de un caso tan trivial como este.

Ante la imposibilidad de probarlo todo, se opta por hacer un conjunto de pruebas que cubran, de manera general, todas las posibles combinaciones que pudieran darse cuando el programa traza alguno de sus posibles resultados. En rigor, no probamos exhaustivamente el programa, pero sí lo suficiente como para asegurar que el software cumple su función, en la inmensa mayoría de casos.

Las pruebas del software son una tarea que se catalogan como críticas, para asegurar que el software cumple su objetivo. Sin ellas no se puede tener la certeza de que el desarrollo del producto se haya completado con éxito. El principal objetivo de las pruebas es encontrar defectos que pudiera tener el software y así corregir los fallos que impidan su correcto funcionamiento. Pero, ¿por qué un software en desarrollo tiene fallos siempre?

Llegados a este punto hay que hacer una distinción entre errores, fallos y defectos.

Los programadores que desarrollan un software son humanos. Ellos, sin ser conscientes, pueden introducir **errores** en el código del programa y en la mayoría de los casos no es trivial solucionarlos, puesto que puede ser que se desencadene una vez de cada mil ejecuciones. Cuando un error salta en ejecución, se produce un **fallo** del sistema. El usuario aprecia que el software no ha cumplido su función satisfactoriamente y es aquí cuando se produce el **defecto**, que es una desviación del resultado esperado. Los defectos son detectados por los "*testers*", (probadores del sistema) a diferencia de los fallos que son detectados por el usuario final. Existe la posibilidad de que un defecto no corresponda a un fallo: si un programa nunca ejecuta el trozo de código defectuoso, nunca se producirá el fallo (8).

Para probar el software, desde un elemento concreto hasta todo el producto en su conjunto, existe un denominador común, una serie de características similares que todas las pruebas tienen, o por lo menos, deberían tener.

- La primera característica clave es la más importante: **encontrar errores**. Es el objetivo de las pruebas y por ello, todas deberían tener una alta probabilidad de encontrar errores.

- La segunda es tener en cuenta **los recursos** ya que, al no ser infinitos, las pruebas no deberían ser redundantes. Por lo tanto, podemos decir que lo que buscamos es efectividad, y así optimizar los recursos de la organización.
- Y, por último, una buena prueba debería tener una **complejidad adecuada**, es decir, la prueba no tiene que ser ni excesivamente compleja ni demasiado simple. Entre estos dos extremos hay que encontrar una solución adecuada.

Una vez vistas las características deseables de una prueba, se estudia a continuación los diferentes tipos de pruebas existen.

Tradicionalmente se pueden agrupar en dos, descubriendo cada una, diferentes tipos de errores: pruebas de caja negra y pruebas de caja blanca (7).

2.4.1. Tipos de Pruebas

La denominación de los tipos de pruebas en el mundo de la informática no es única y se puede decir que, dependiendo de la fuente, cambian desde los nombres de las pruebas hasta la clasificación de las mismas. Existen varios tipos de clasificaciones diferentes: clasificación según quien las ejecuta, según qué hacen, según su funcionamiento etc.

En este trabajo la clasificación será la presentada en la ilustración 3 donde encontramos una distinción, por el tipo de prueba, de caja negra o de caja blanca. Dentro de las pruebas de caja negra, existen dos subtipos más dependiendo de la persona encargada de lanzar la prueba, si es un desarrollador o un tester, o bien, si es el cliente.

En las pruebas de caja blanca no es necesario distinguir la persona que ejecutará estas pruebas puesto que siempre será alguien del equipo de desarrollo. En la siguiente ilustración está descrita gráficamente la clasificación para mejorar la compresión de esta.

Ilustración 3: Tipos de Test

2.4.2. Pruebas de caja negra

Las pruebas de **caja negra** son aquellas que comprueban que cada función del programa es operativa, de forma que por cada funcionalidad se prueba si: los parámetros de entrada son correctos, si al pasar por la función toda la información mantiene su integridad y si finalmente los resultados de salida son adecuados. Por lo tanto, estamos comprobando si los requisitos funcionales son correctos. En estas pruebas no se tiene acceso al código fuente: el interior de la "caja" es opaco e inaccesible. Normalmente este tipo de pruebas son un complemento a las de caja blanca.

Además de la clasificación propuesta anteriormente, existe otro punto de vista para clasificar las pruebas de caja negra: el punto de vista de la técnica, y se compone de cuatro tipos de pruebas. No se profundizará en cada una de ellas, puesto que se seguirá la clasificación propuesta, pero si se mencionará: la primera de ellas son las pruebas de **partición equivalente**, útil para reducir el número de casos de prueba a desarrollar, la segunda la forman las pruebas de **análisis de límites**, que es complementario al grupo anterior de pruebas, la tercera son las **pruebas de comparación**, útil para desarrolladores que utilizan sistemas redundantes y finalmente tenemos el grupo de

pruebas en **entornos especiales** que, a su vez, se divide en pruebas para sistemas de tiempo real, arquitecturas cliente-servidor e interfaces gráficas.

Siguiendo la clasificación establecida al comienzo de este apartado se describirán, en primer lugar, las pruebas ejecutadas por el cliente (Test de Aceptación y Funcionales) y después las pruebas ejecutadas y programadas por los testers del equipo de desarrollo del software (Test de Sistema, de Regresión y de Humo).

a) Pruebas de Aceptación

Los test o pruebas de aceptación (TA) se realizan en la fase posterior a las pruebas del código fuente, puesto que el objetivo de estas pruebas es hacer un test global al artefacto que se va a entregar al cliente. En un TA se crea un escenario simulando al usuario final, y se prueba si los requisitos cumplen con las expectativas del cliente. En este tipo de pruebas no es necesario el acceso al código fuente, sólo se prueba lo acordado en el escenario, para comprobar que lo que sucede es lo esperado. En desarrollos de software que son complejos y que tienen un gran número de requisitos, es posible que se ejecuten varios test con escenarios diferentes, en función de las necesidades del software, y que se prueben varios escenarios para una misma parte del software, variando los eventos, desde los más comunes a los casos más extremos e improbables.

Los TA son ejecutados tanto por los desarrolladores, antes de la entrega al cliente, como por el cliente. Por ejemplo, hay metodologías ágiles que al acabar un sprint los desarrolladores ejecutan pruebas de aceptación antes de entregar el artefacto al cliente. Si es el caso en el que el cliente está ejecutando un TA, se recoge el *feedback* y se continua con los siguientes pasos, en función del método de desarrollo que se esté usando.

Las herramientas más utilizadas para ejecutar pruebas de aceptación son:

- FIT (9)
- FitNesse (10)
- Parasoft Service Virtualization (11)

b) Pruebas Funcionales

Un test o prueba funcional es aquella que comprueba el flujo de información que entra, y la salida que se obtiene de un software. Para ello es necesario basar el test en los

requisitos del software que se está probando. Las pruebas funcionales derivan de los test de aceptación, puesto que lo que se comprueba es una funcionalidad con valor de negocio. Un test funcional siempre será un test de aceptación, pero esta propiedad no es conmutativa, un test de aceptación no tiene porqué ser funcional.

Existen fundamentalmente tres tipos de pruebas funcionales: de compatibilidad, regresión y de humo. Existen otros tipos de prueba funcional que no se verán en este trabajo, como las pruebas exploratorias o de integración.

En función del tipo de ejecución existe otra clasificación para las pruebas funcionales: manuales o automáticas. Esta clasificación tiene importancia para el *agile testing*, ya que una de las prácticas de las pruebas ágiles es intentar ejecutar las pruebas automáticamente para reducir el tiempo en el desarrollo de un software. Esto no quiere decir que se prohíba el uso de pruebas manuales, de hecho, es recomendable su uso, pero sí que existe una tendencia a imponer las pruebas automáticas frente a las manuales.

Las **pruebas manuales** son test que desarrolla y ejecuta un miembro del equipo de desarrollo. Una prueba manual es una prueba de sistema y también una prueba de aceptación.

Es habitual que las herramientas para crear test manuales estén integradas en los IDE (Integrated Development Environment) más populares, pero también existen herramientas específicas como:

- QTest
- PractiTest
- TestFLO for Jira (12)
- XQual (13)

Las **pruebas automáticas** son test que ejecuta el IDE u otro software específico sin necesidad de estar paso a paso ejecutando el test manualmente. Las pruebas automáticas permiten al programador continuar codificando sin necesidad de pararse a crear una prueba manual. El IDE prueba la ejecución del código de forma automática, sólo hay que especificar algunos datos como la salida que deben dar los métodos o las funciones. Esto permite un testing continuado a lo largo de todo el proceso y es la esencia del *agile testing* (se verá más adelante en el capítulo 5).

Existen herramientas específicas para crear pruebas automáticas. Las más famosas de código abierto y utilizadas son:

- FitNesse
- Tosca (14)
- Codacy
- Mocha (15)
- Sahi (16)
- TestComplete (17)
- Appium (18)

Existen herramientas comerciales para aplicar este tipo de pruebas como:

- HP UFT (19)
- IBM Rational Test Workbench (20)
- Ranorex UI (21)
- Seapine QA Wizart Pro (22)
- Robot (23)
- SmartBear Test Complete (17)
- Original Software (24)
- TAF (25)
- OpKey (26)
- Sahi PRO (16)

c) Pruebas No Funcionales

Las pruebas no funcionales son pruebas creadas para medir tanto las características relacionadas con el comportamiento del sistema como su rendimiento, estrés ante tráfico o carga masiva de datos, fiabilidad, etc. Dentro de este tipo de pruebas existen varios tipos en función de la característica a analizar: pruebas de carga, de estrés, de fiabilidad, de usabilidad o de rendimiento. Este tipo de pruebas son realizadas por el equipo desarrollador.

Existen muchos tipos de herramientas dada la gran cantidad de tipos de prueba que hay en esta categoría. A continuación, se mencionan las herramientas más completas o las más famosas de código abierto:

- Apache JMeter (27)
- Gatling (28)
- Locust (29)
- Rational Performance Tester (30)
- WAPT (31)
- Testing Anywhere (32)
- Appvance (33)
- QEngine (Manage Engine) (34)
- OpenSTA (35)

Entre las herramientas comerciales destacan por su uso:

- Neotys (36)
- SOASTA CloudTest (37)
- Parasoft Load Test (38)
- HP PC & Load Runner (39)
- BlazeMeter (40)
- Compuware Gomez and dynaTrace (41)

d) Pruebas de Compatibilidad

Las pruebas de compatibilidad son aquellos test que consisten en ejecutar el software en varias plataformas y comprobar que tenga un comportamiento idéntico en todas. Estas pruebas se ejecutan para comprobar que el software es portable y que no existen deficiencias a pesar del cambio del entorno. Está enfocada a proyectos destinados a varias plataformas, como por ejemplo las aplicaciones móviles, y pueden ser ejecutadas tanto por el cliente, en fases finales o intermedias, como por los programadores. Estas pruebas se pueden automatizar, pero generalmente son manuales puesto que automatizando este tipo de pruebas se pierde eficacia en la detección de errores.

En el caso de los navegadores, por ejemplo, las herramientas más usadas para ejecutar pruebas de compatibilidad son:

- BrowserStack (42)
- TestingWhiz (43)
- Cross Browser Testing (44)
- Rapise (45)

e) Pruebas de Regresión

Los test de regresión son pruebas que intentan detectar errores cuando una funcionalidad o un módulo del software ha sido modificado, y ver cómo ha cambiado el comportamiento del software de manera general con una modificación puntual. Este tipo de errores se producen generalmente por no llevar una política de control de versiones adecuada. Por lo tanto, el objetivo de este tipo de pruebas es comprobar que las modificaciones en el software no han introducido fallos.

Este tipo de pruebas es lanzado por los desarrolladores o los tester, y es posible su automatización.

Para este tipo de pruebas se utilizan las mismas herramientas que se usan en las pruebas unitarias y en las de sistemas, pero también tienen herramientas especializadas de código abierto como:

- Winrunner (46)
- Canoo (47)
- VTest (48)
- AdventNet QEngine (34)

Otras herramientas comerciales para aplicar este tipo de pruebas son:

- HP UFT (49)
- IBM Rational Test Workbench (20)
- Ranorex UI (21)
- Seapine QA Wizart Pro (13)
- SmartBear Test Complete (17)
- SOASTA (37)
- OpKey (26)

f) Pruebas de Humo

Los test de humo son pruebas sencillas y cortas que comprueban de forma individual partes básicas del sistema. Estas pruebas son lanzadas por el tester en la fase de desarrollo o codificación del software. Al igual que las pruebas de regresión y de sistema, es posible su automatización, pero no es obligatoria.

Para este tipo de pruebas, al igual en las pruebas de regresión, se utilizan las mismas herramientas que se usan en las pruebas unitarias y en las de sistemas. También hay herramientas que no son específicas, pero son muy usadas para ejecutar pruebas de humo:

- Maven (Easy Mock) (50)
- Cargo (51)

g) Pruebas de Sistema

Una prueba de sistema es un test de integración, con la diferencia de que no se puede tener acceso al código para formalizar la prueba. Ahora todo depende del escenario creado para ésta y de los resultados externos que emite el software. Este tipo de pruebas las ejecuta un tester y se deben ejecutar en la fase de prueba. Es posible su

automatización, pero en algunos casos es posible que lanzar la prueba manualmente sea más eficaz.

Las herramientas de código abierto más utilizadas para ejecutar este tipo de pruebas son:

- Selenium (52)
- Fit (10)
- WET (53)
- Watir (54)

h) Pruebas de Rendimiento

Las pruebas de rendimiento son test que testean el software en situaciones extremas (55). Existen varios subtipos de pruebas dentro de estas entre las que encontramos:

a) Pruebas de Carga: Prueban el volumen máximo de datos que el software es capaz de persistir en memoria, tanto volátil como no volátil.

b) Pruebas de Estrés: Someten al software a peticiones masivas para comprobar que el software responde correctamente ante situaciones de tráfico intenso de datos.

c) Pruebas de Estabilidad: El software es sometido a pruebas con otros programas para comprobar que su interacción en memoria no ponga al producto que se está evaluando ante una situación de inestabilidad.

d) Pruebas de Picos: El producto es sometido ante picos puntuales de carga y estrés intenso para probar la capacidad de escalado del software ante punto intensos de tráfico.

2.4.3. Pruebas de caja blanca

Las pruebas de **caja blanca** son aquellas pruebas que se centran en analizar el funcionamiento interno del programa para ver si se ajusta a las especificaciones y se comporta correctamente. Por lo tanto, ahora sí se tiene acceso al código fuente, la "caja" es blanca y se puede ver su interior.

Este tipo de pruebas son más minuciosas que las de caja negra, puesto que se comprueban todos los caminos lógicos del software (bucles, saltos, condiciones…) y

se examina el estado del programa en varios puntos. El problema de las pruebas de caja blanca es que se tardaría muchísimo tiempo en analizar todo el código y, en un desarrollo software, el tiempo es un recurso acotado a las limitaciones del proyecto. Lo habitual es comenzar por ejecutar pruebas de caja blanca y ya en fases posteriores seguir con las pruebas de caja negra.

Al igual que en las pruebas de caja negra, en este tipo de pruebas, existe otra clasificación adicional, que no seguiremos, en función de la técnica usada: pruebas del **camino básico**, aplicadas tras la implementación de cada módulo, y **las pruebas de las estructuras de control**, donde se probarán bucles y sentencias condicionales.

Siguiendo la clasificación establecida para este TFG mencionada en la ilustración 3, los principales tipos de pruebas de caja blanca son: las pruebas unitarias y las pruebas de integración. Ambas estarán siempre automatizadas por alguna herramienta software, y por supuesto, estarán ejecutadas por algún miembro del equipo desarrollador nunca por el cliente (puesto que, al cliente, no le interesa los detalles técnicos sólo la funcionalidad del producto en su conjunto).

a) Pruebas Unitarias

En una prueba unitaria se prueba la parte más pequeña de un software: los módulos, las clases, los métodos etc. En este tipo de test, se ejecutan pruebas de cada parte del código de manera independiente. Son test básicos que hay que lanzar para comprobar que las rutinas del código cumplen con lo esperado. Las condiciones del entorno de la prueba son diseñadas por los desarrolladores, que también realizan la tarea de verificar los resultados obtenidos. Estos test son básicos y suponen el primer pilar para llegar a la calidad del software, son pruebas obligatorias y no se pueden eludir. Una buena prueba unitaria, reúne una serie de características comunes que deben tener.

Deberían poder ser ejecutadas de manera automática y no manual. Automatizar las pruebas con herramientas de software ayuda a mejorar la productividad del tester o del desarrollador.

Otra característica deseable es tener la capacidad de poder repetirlas tantas veces como el desarrollador contemple, lo que lleva a: por una parte, tener tiempo suficiente, para ejecutar las pruebas y quedar satisfecho con los resultados y por otra, la velocidad en las pruebas. De estos dos factores, el desarrollador sólo puede influir en una, la

velocidad, puesto que el tiempo de pruebas lo marca otro miembro del equipo. El desarrollador debe optimizar los test unitarios para que sean rápidos y eficaces.

Es preciso que se pruebe la totalidad del código con este tipo de pruebas. Si no se cumple esto, habrá una parte del software que estará sin probar y la probabilidad de resultados inesperados en la ejecución, será exponencialmente más alta, que si probamos todo o casi todo el código.

Las pruebas unitarias tienen que ejecutarse aisladamente unas de otras y está claro que la ejecución de un test no debe influir en otro.

Estas características que se han descrito están recogidas en el principio FIRST (Fast, Isolation, Repeteable, Small y Transparent). Si no cumplen estas propiedades existe el riesgo de que un test unitario no sea válido.

Las herramientas de código abierto más conocidas y utilizadas para ejecutar este tipo de pruebas son:

- NUnit (56)
- JUnit (57)
- TestNG (58)
- DBUnit (59)
- Sahi (16)
- XUnit (60)
- Jmock (61)
- Cgreen (62)

Las herramientas comerciales más famosas son:

- Parasoft C/C++ TEST (63)
- Typemock (64)
- AgitarOne (65)

b) Pruebas de Integración

Los test de integración son pruebas que tienen una filosofía parecida a la de los test unitarios. La diferencia reside en que, mientras en un test unitario estamos probando una parte muy básica del código y teniendo en cuenta los principios FIRST, en las pruebas de integración se rompe con esas reglas FIRST, abarcando más código, y con ello, es posible probar varias partes del sistema a la vez. Su propio nombre indica el objetivo de éstas: integrar varias partes del sistema para hacer la prueba.

Los tipos de errores que detecta este tipo de pruebas suelen ser dos: problemas de programación entre distintas partes del software y problemas de interoperabilidad.

- **Los problemas entre las distintas partes del software:** surgen cuando varios módulos terminan por crear un conflicto en datos o valores del software, ya sean valores de salida o valores usados por el programa. Para resolver este problema se siguen primitivas de control para la concurrencia de los datos, como por ejemplo los semáforos, que dan lugar a errores como el interbloqueo[7] o el bloqueo activo[8], o los monitores, mucho más eficaces que estos.
- El **problema de la interoperabilidad:** se da en varios niveles.
    a) A nivel de sistema, cuando el software no es compatible con el sistema operativo o las librerías de una determinada versión.
    b) A nivel del lenguaje de programación, cuando el software no es compatible con otros lenguajes de programación.
    c) A nivel de protocolo, cuando el sistema interactúa con métodos y no están coordinados por discrepancias en el protocolo de intercambio de información.

Para resolver este tipo de problema las pruebas de integración siguen dos estrategias distintas:

- Incrementales: se empiezan probando pocas partes del sistema y vamos incrementando el número de partes hasta probarlos todos, o
- No incrementales: cuando se prueba todo el software de golpe como un todo.

Dentro de la estrategia de pruebas de integración incrementales hay varios tipos: incrementos en profundidad, en anchura o descendentes.

---

[7] Interbloqueo: *"es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente que compiten por recursos del sistema o bien se comunican entre ellos"* (113)

[8] Bloqueo activo: *"es similar a un interbloqueo, excepto que el estado de los dos procesos envueltos en el bloqueo activo constantemente cambia con respecto al otro".* (114)

Para ejecutar este tipo de pruebas, además de herramientas dirigídas únicamente a las pruebas de integración, también se pueden usar herramientas de pruebas unitarias. Las herramientas especializadas en este tipo de pruebas más famosas son:

- HttpUnit (66)
- SoapUI (67)
- POSTMAN (68)
- RESTClient (69)

2.4.4.Pruebas de caja gris

Existe otro tipo de test llamados pruebas de **caja gris**, que es un hibrido compuesto por las pruebas de caja negra y de caja blanca. Se usa en casos especiales en los que es necesario tener en cuenta el código fuente y a la vez saber los valores de salida de los módulos. Se suele usar poco porque lo más frecuente es dividir en los dos tipos anteriores las pruebas.

Hay herramientas para este tipo de pruebas, pero no son exclusivas de las pruebas de caja gris. Las más conocidas son:

- Selenium (52)
- Appium (70)
- Rational Functional Tester (20)
- Chrome Dev Tools (71)

2.5. CONCLUSIÓN DEL CAPÍTULO

En este capítulo se ha dado un repaso de las características fundamentales del desarrollo de software, el modelo clásico del ciclo de vida, como ejemplo genérico, y se han comentado los aspectos fundamentales del control de calidad del software y de la mayoría de los diferentes tipos de pruebas de software que existen.

Una vez repasados los conceptos que se necesitan, podemos continuar con las metodologías ágiles. Se observará la evolución y los motivos que llevaron a la creación de este tipo de métodos para desarrollar software. Y posteriormente se estudiará dos tipos de metodologías concretas pertenecientes a este modelo ágil de desarrollo para entender mejor su funcionamiento.

3. DESARROLLOS ÁGILES

En este capítulo se estudia la creación y evolución de las metodologías ágiles. También se explica el funcionamiento de dos metodologías ágiles SCRUM y XP (eXtreme Programming).

El aumento de la complejidad de los sistemas software, ha supuesto la necesidad de aumentar la calidad de los productos software. Ahora el cliente espera mucha más calidad en el software que hace unas décadas. Noaroki Kano fue un profesor japonés especializado en la calidad de productos (72) que en los años 70 publicó herramientas para crear prioridades en las necesidades y atributos que aportaban más satisfacción al cliente. Estableció tres tipos de prioridades que estaban asociadas directamente con la satisfacción esperada del cliente (Ilustración 4).

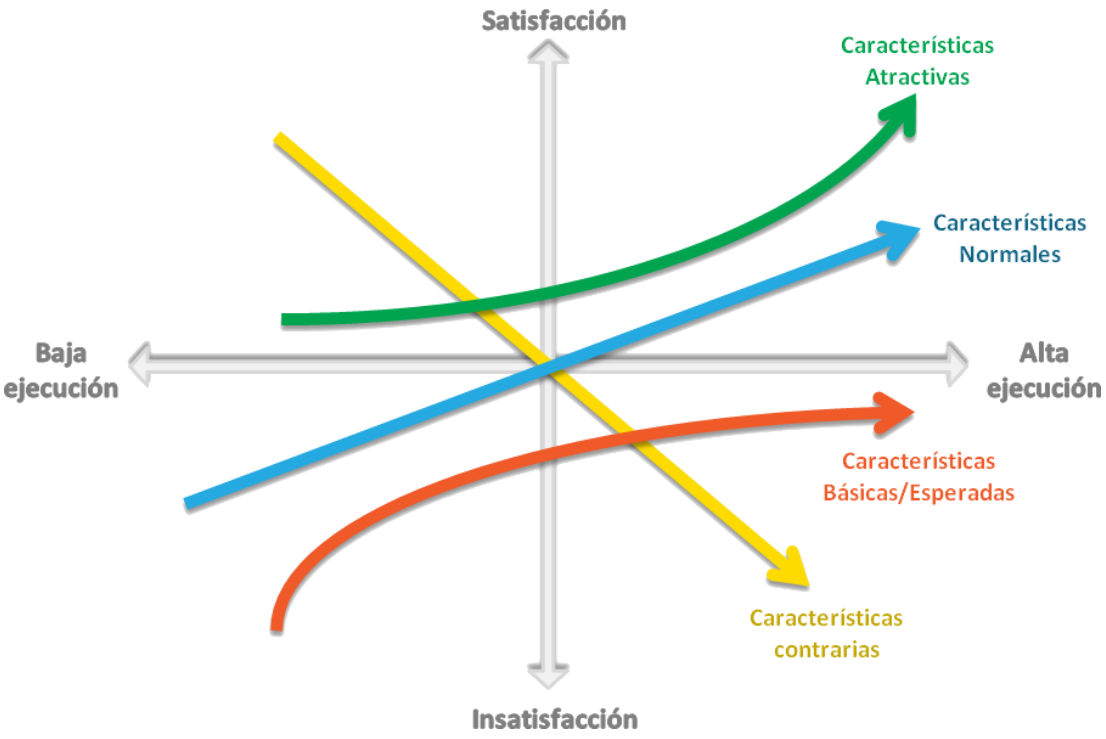

Ilustración 4: Diagrama de Kano (73)

1) Las **características básicas o necesidades básicas:** son aquellas características básicas necesarias para que el software cumpla su función correctamente. Sin ellas, no sólo se produce un escenario de falta de calidad, sino que el desarrollo sería un fracaso absoluto.

2) Las **características normales o necesidad de prestaciones:** que son las necesidades que el cliente puede medir y cuantificar para comparar con la competencia.

3) Las **características atractivas o necesidades de entusiasmo**: representa aquellas características que satisfacen necesidades no previstas por el cliente y hacen que el software sea distinto o superior al software de la competencia. Con éstas, el cliente se muestra entusiasmado y puede llegar a fidelizarse con la organización desarrolladora del software.

Para entender mejor el diagrama de la ilustración 4, se propone el ejemplo de los procesadores de texto. Imagine que tenemos una empresa que desarrolla software y existe un cliente que pide desarrollar un procesador de textos, como puede ser el WordPad, Libre Office, Word etc. El cliente sabe lo que quiere, pero de una manera difusa, y por ello no sabe qué características debe tener el software, una situación típica. Así que, utilizando una metodología concreta, se empieza el desarrollo de su software.

Siguiendo el diagrama de Kano, en función del número de características a desarrollar, encontramos las siguientes situaciones:

Se desarrolla un procesador de texto que cubre las necesidades básicas, que permita escribir un texto, guardarlo en memoria no volátil, imprimirlo, etc. Cubre las necesidades del cliente, pero es entendible que aun dando al cliente lo que está pidiendo, no esté contento porque es muy básico.

Sin embargo, si durante el análisis de los requisitos se descubren características deseables (necesidad de prestaciones) y se desarrollan funciones como dar formato al texto, inserción de imágenes, corrector ortográfico..., el cliente estará mucho más contento, puesto que el software que ha pedido cumple con prestaciones suficientes como para que pueda competir con la competencia.

Si además de todo lo anterior, se desarrolla junto a las demás características, otras funcionalidades como: almacenamiento en la nube, guardado de archivos multiplataforma o posibilidad de trabajar concurrentemente con un archivo. El cliente, en este caso, estará entusiasmado, puesto que el software tiene características que sobresalen a lo esperado y compiten fuertemente en el mercado.

Con los diagramas de Kano se extrapola el factor del individuo a la calidad de software. El cliente, es el primero que debe estar contento con su producto, y requiere toda la atención. Pero ¿qué tiene que ver la importancia del individuo y su satisfacción con los métodos agiles? La respuesta está en la definición formal del termino ágil y en cómo aparece en la historia del software.

Un **desarrollo ágil** está formado por una serie de pautas usadas en el desarrollo de proyectos de software que tienen una duración entre un corto y mediano espacio de tiempo. La duración suele corta, como máximo un año, no teniendo una duración mínima concreta.

En el año 2001 diecisiete expertos se reunieron para tratar sobre técnicas y modelos para desarrollar software. Es en ese momento cuando se definió la metodología ágil y se publica el famoso "manifiesto ágil".

3.1. EL MANIFIESTO ÁGIL

En este apartado, se presenta el manifiesto original de 2001 que resultó de la reunión en Utah (74). El objetivo era poner en práctica valores y principios para que las organizaciones pudieran desarrollar software de manera rápida y con una capacidad de respuesta a cambios, eficaz.

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

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

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.

Los autores del manifiesto ágil son los siguientes[9]:

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

Tras deducir los cuatro valores del manifiesto, los autores firmaron los doce principios que componen el manifiesto:

1. *Nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor.*
2. *Son bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo. Los procesos ágiles se doblegan al cambio como ventaja competitiva para el cliente.*
3. *Entregar con frecuencia software que funcione, en periodos de un par de semanas hasta un par de meses, con preferencia en los periodos breves.*
4. *Las personas del negocio y los desarrolladores deben trabajar juntos de forma cotidiana a través del proyecto.*
5. *Construcción de proyectos en torno a individuos motivados, dándoles la oportunidad y el respaldo que necesitan y procurándoles confianza para que realicen la tarea.*
6. *La forma más eficiente y efectiva de comunicar información de ida y vuelta dentro de un equipo de desarrollo es mediante la conversación cara a cara.*
7. *El software que funciona es la principal medida del progreso.*

[9] Son los 17 autores del Manifiesto Ágil

8. *Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante de forma indefinida.*
9. *La atención continua a la excelencia técnica enaltece la agilidad.*
10. *La simplicidad como arte de maximizar la cantidad de trabajo que no se hace, es esencial.*
11. *Las mejores arquitecturas, requisitos y diseños emergen de equipos que se auto-organizan.*
12. *En intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo y ajusta su conducta en consecuencia.*

Con este documento se da mucha importancia, por primera vez, a la necesidad de sobreponer las personas al proyecto, es decir, enfatizar las relaciones entre los desarrolladores y el cliente, y favorecer la gestión de recursos en equipos pequeños.

Kano vio en 1970 esa relación entre el cliente y el producto. Colaborar con el cliente es absolutamente necesario.

Normalmente, como se argumenta en el CHAOS Report (5) uno de los motivos por el cual un proyecto fracasa es debido a la poca colaboración con el cliente. Es cierto que hay que establecer unos límites contractuales para marcar los derechos y obligaciones del cliente y del desarrollador, pero se puede perder la conexión con el cliente, y esto terminará provocando que, al final, el cliente no esté satisfecho con el producto porque no era lo que quería.

Otra novedad introducida por las metodologías ágiles es que por primera vez se da más importancia a que el software sea funcional, en vez de crear mucha documentación. Esto no quiere decir que no sea necesario documentar el proyecto, se refiere a no estar obsesionado con ello y a que la documentación no sea excesiva. Lo importante es centrarse en que el software funcione.

El objetivo es crear un software funcional, de forma que se vayan creando pequeños artefactos todos los días o en un margen de tiempo pequeño, semanal o mensualmente. Esto se hace así porque se valora más la respuesta a los posibles cambios que puedan surgir durante el proyecto, que una metodología pesada. Una metodología ágil asume

bien los cambios, pero se tiene que ser riguroso en este aspecto, puesto que hay que conocer el límite de modificaciones que pueda aguantar el proyecto software. Hay cambios que no son asumibles, aunque el modelo esté predispuesto a las modificaciones.

En la tabla 1 se ve con mayor claridad las diferencias con respecto a las metodologías tradicionales (75):

Tabla 1: Comparativa entre Métodos Ágiles y Métodos Tradicionales

DESARROLLO ÁGIL	DESARROLLO TRADICIONAL
Preparados para afrontar cambios durante el desarrollo del proyecto	Cierta resistencia a cambios
El cliente es parte fundamental del proyecto	El cliente interactúa con los desarrolladores mediante reuniones
Grupos pequeños trabajando en el mismo sitio	Grupos grandes y distribuidos
Software funcional	Documentación extensa y rigurosa
Proceso con pocos principios	Proceso muy controlado, muchos principios y normas
Procesos solapados y simultáneos	Procesos lineales y secuenciales
Prioridad al código	Prioridad a la arquitectura de los sistemas

Los resultados entre una metodología y otra se encuentran documentados y cuantificados (76). Existe la evidencia de un porcentaje superior de éxito al usar una metodología ágil frente a un modelo tradicional en desarrollos a corto plazo.

A continuación, se estudia un caso: los proyectos realizados usando un método ágil frente a proyectos que fueron desarrollados con una metodología en cascada, que es un modelo de desarrollo tradicional.

Para este caso, el 39% de los proyectos ágiles acaban en éxito frente a un escaso 11% que usa metodologías en cascada. Así lo documentó Standish Group y su famoso CHAOS Report 2015 (5).

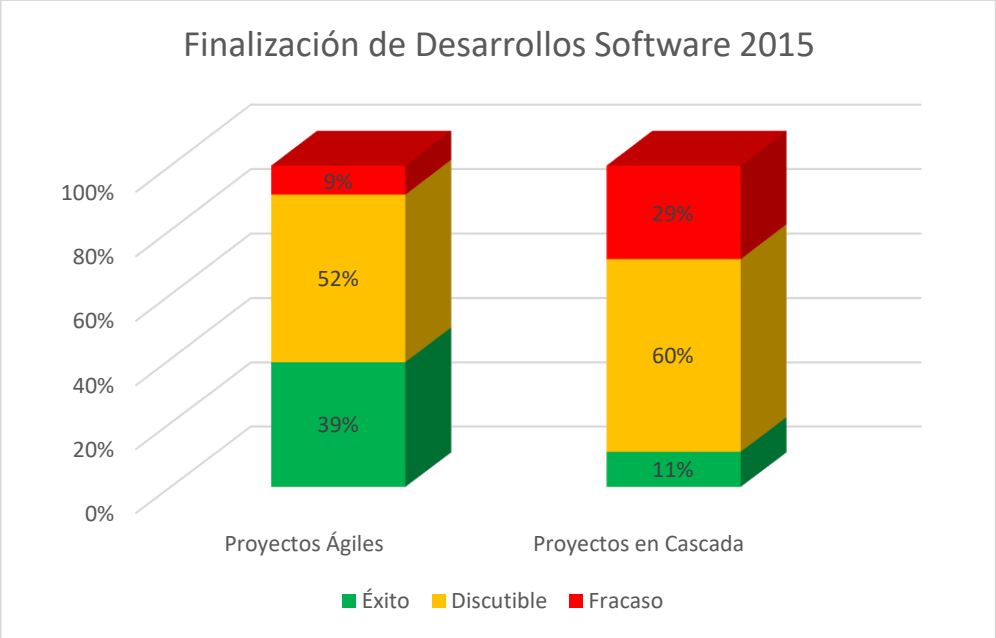

Ilustración 5: Tasa de éxito en desarrolos de software 2015 (5)

El dato más llamativo son las tasas de fracaso. Destaca que sólo el 9% de los proyectos llevados a cabo mediante metodologías ágiles fallaron, frente al elevado 29% de los proyectos con metodología en cascada.

Viendo estos datos se puede afirmar que estadísticamente existen más probabilidades de tener éxito al desarrollar un software desarrollado con un método ágil, que con un desarrollo en cascada.

3.2. LA METODOLOGÍA

Una vez establecidos los principios ágiles y el manifiesto en sí, surgieron varios tipos de metodologías distintas, cada una con sus peculiaridades, pero en general todas tienen características comunes propias de las metodologías ágiles.

Entre los principales desarrollos ágiles existentes se pueden mencionar como los más importantes:

- ASD (Adaptative Software Development)
- Crystal Methodologies
- DSDM (Dynamic Systems Development Method)
- FDD (Feature-Driven Development)
- KANBAN
- LD (Lean Development)
- MSF (Micosoft Solution Framework)
- OSS (Open Source Software Development)
- RUP, AUP y EUP (Rational/Agile/Enterprise Unified Process)
- SCRUM
- XP (Xtreme Programming)

En la actualidad los métodos ágiles más usados son SCRUM y Programación Extrema (XP). No obstante, el desarrollo ágil por excelencia hoy en día es SCRUM, que se describe en el apartado siguiente. Para seguir avanzando se necesita seguir una metodología concreta, y así, no tener en cuenta tanto la teoría e ir profundizando en la practicas, las técnicas, herramientas y los procedimientos reales que están siendo usando hoy en día. En este TFG se centra en estudiar Scrum, por ser la metodología más utilizada, y XP, por ser la metodología precursora del *agile testing*.

3.2.1.SCRUM

En el año 1986 dos académicos japoneses (Nonaka y Takeuchi) publicaron un artículo llamado "The New Product Development Game" (77) en el cual mostraban las prácticas de grandes empresas japonesas innovadoras, que conseguían realizar productos de características similares a los de su competencia, en menos tiempo, con menos costes, y con más calidad. En dicho artículo, estos académicos hicieron una comparativa entre la forma de trabajar en equipo de las empresas japonesas exitosas, con el avance en formación de un tipo de jugada de Rugby (scrum en inglés), a raíz de lo cual quedó acuñado el término "scrum" para referirse a ella.

"El enfoque de carrera de relevos en el desarrollo de productos, puede entrar en conflicto con los objetivos de máxima velocidad y flexibilidad. En su lugar un enfoque holístico o estilo rugby, donde un equipo intenta recorrer la distancia como una unidad, pasando la pelota hacia adelante y hacia atrás, puede servir mejor a los actuales requisitos competitivos" (77)

Esto es el preámbulo al término Scrum que se adoptaría formalmente en 1995 por los ingenieros de software Ken Schwaber y Jeff Sutherland (78). Su adopción se debe a que, en Scrum, se realizan un conjunto de buenas prácticas entre las que destaca trabajar en equipo y así obtener el mejor resultado en un proyecto, al igual que la juagada "scrum" en el Rugby.

Scrum combina, los modelos de procesos iterativos con los valores del manifiesto ágil, de forma que los equipos que desarrollan el producto, tienen las siguientes características: Incertidumbre, Gestión de Organización, Control y Transmisión de conocimiento.

- No tener estrictamente detallado el plan para llevar a cabo un proyecto software, introduce cierta **incertidumbre** al equipo de desarrollo. Se planifica el objetivo, pero no cómo llevarlo a cabo detalladamente. Este estado de tensión es adecuado para la motivación de los equipos de desarrollo, que debería servir también para darles cierta motivación y espacio. La planificación del proyecto es adaptativa, donde se puede distribuir la carga de trabajo por semanas (más o menos tiempo en función del objetivo marcado por los sprint, que en ningún caso debe superar un mes).
- La **gestión de la organización** se refiere a la capacidad de que los equipos se gestionen adecuadamente a sí mismos (autogestión). Nadie tiene una jerarquía de mando sobre el equipo. Como mucho se lidera al equipo, se facilita tareas de integración y se resuelven problemas. No obstante, es evidente que en una organización hay jefes, que tienen la potestad de decidir planes de contratación de personal, control del cumplimiento de objetivos y tareas, realización planes relacionados con la gestión del presupuesto o asignar tareas a diferentes trabajadores. Pero este tipo de liderazgo es para la gestión de la organización en su conjunto, y no para gestionar un equipo de desarrollo. Scrum sólo gestiona proyectos software, no tiene en cuenta los recursos de la empresa o cualquier otro aspecto que no esté estrictamente dentro del desarrollo de un software.
- Se debe generar un ambiente de igualdad entre todos los trabajadores del grupo, para que todos desarrollen su capacidad de ser creativos. Por lo tanto, el **control** aplicable en un grupo debería ser moderado.

- La última característica se refiere a compartir **conocimientos** entre todos los integrantes del grupo. Todos aprenden de todos.

Estos equipos deben estudiar los requisitos, creando una lista de prioridad de características y capacidades que debe tener el producto. El cliente va a ver cómo evoluciona el prototipo de manera incremental, puesto que las entregas son iterativas y en periodos cortos y fijos. El cliente deberá dar el visto bueno, o no, de cada paso que los equipos den en cada iteración.

Lo ideal para un equipo de Scrum es que esté formado entre cinco y nueve personas. Si se supera este número, se deberá usar una jerarquía con varios equipos.

Una vez descrita la filosofía a aplicar, en los apartados siguientes, se estudia los pilares fundamentales en los que está basado este método: los roles, los artefactos, las reuniones y los Sprint.

a) Los Roles en Scrum

Los roles son los diferentes papeles que juegan los integrantes de un equipo de desarrollo. Los roles principales son: el dueño del producto (Product Owner), el líder del equipo (Scrum Master) y el miembro del equipo (Development Team Member).

- El **Product Owner:** es el rol que establece el enlace entre el cliente (stakeholders) y el equipo. Esto se hace así, puesto que existe la posibilidad de que el cliente no pueda estar colaborando con el equipo de desarrollo a tiempo completo. Con este rol se pone expresamente de manifiesto la importancia que Scrum da al contacto con el cliente. La persona encargada de desarrollar este rol es responsable de la visión del producto y de la gestión económica, decidiendo las características que hay que desarrollar en cada sprint y de su validación.
- El **Scrum Master:** ayuda a mantener los valores de Scrum en el equipo de desarrollo, realizando labores de liderazgo. Es el encargado del Product Backlog (Pila del Producto) y asigna el trabajo que debe realizar los miembros del equipo en el sprint.
- Los **miembros del equipo:** deben diseñar, implementar y validar el sistema que están desarrollando, aplicando los valores Scrum inculcados por su líder.

Existen otros roles comunes en un equipo ágil con Scrum, como los arquitectos o los administradores de sistemas, pero no se tratarán aquí, puesto que no son relevantes y asumen competencias de proyectos más complejos. Para formar un equipo ágil, estándar, es suficiente con los roles que se acaban de describir. Incluir en un equipo Scrum estos roles, no cambia ni la filosofía ni la mecánica de trabajo. Simplemente se unirían a los miembros del equipo, pero con competencias exclusivas de sus campos.

b) Los Artefactos

Los artefactos son todo tipo de productos tangibles resultantes de un proceso de software, necesarios para entender el desarrollo del mismo. En general, pueden ser artefactos: el código del programa, los casos de usos, los planes del proyecto, los riegos, etc. En lo referido a Scrum, los principales artefactos resultado de la propia dinámica de este desarrollo son: La Pila del producto (Product Backlog), Sprint Backlog y el Gráfico de quema del Sprint (Sprint Burn Down Chart).

- La **Pila del Producto (Product Backlog):** es un artefacto que prioriza las tareas a efectuar en el sprint por su valor de negocio. En la pila se muestra el esfuerzo o coste de desarrollo de cada tarea, usando una puntuación cualitativa que estime oportuna el equipo (story points) (79). El responsable es el Product Owner, que representa la voz del cliente en forma de requisitos funcionales. Como la pila es un ente dinámico, a medida que avanza el desarrollo las tareas se vuelve a priorizar. Hay que tener en cuenta que los requisitos pueden cambiar a lo largo del proyecto.
- El **Sprint Backlog:** se crea al comienzo de cada sprint (más adelante cuando se procede a estudiar el funcionamiento de Scrum, se explica qué es un sprint). Consiste en seleccionar de la pila del producto las características a implementar en el sprint. Ahora las tareas se miden en horas de trabajo. El documento se crea en una reunión entre el equipo de desarrolladores, el Product Owner, el Scrum Master y el cliente. Este documento no se modifica hasta que no termine el sprint.
- El **Sprint Burn Down Chart:** es una gráfica que representa la evolución del proyecto, mostrando los requisitos que quedan por realizar al comienzo de cada sprint. En ella se puede ver la duración que le queda al proyecto para llegar a su fin. Es una técnica muy usada también en otras metodologías.

c) Las Reuniones

Las reuniones son organizadas por el líder del proyecto (Scrum Master). Existen varios tipos dependiendo de la fase concreta en la que se encuentre el proyecto. Por lo general debe haber como mínimo: una reunión de planificación del Sprint (Spring Planning Meeting), una reunión Diaria (Daily Scrum o Stand-up Meeting) y una reunión de Revisión del Sprint (Spring Review Meeting).

- La **reunión de planificación del sprint:** es una reunión donde se reúnen el Scrum Master, el Product Owner, el equipo de desarrolladores y el cliente. En ella se ponen de acuerdo para ver qué características hay que desarrollar en el próximo sprint. El resultado de esta reunión es el sprint Backlog.
- La **reunión diaria:** es una reunión de corta duración (suele ser de 15 minutos, una forma de garantizarlo es que sea de pie) para sincronizar el trabajo, ver los objetivos o plantear problemas (que no se resuelven en esta reunión).
- La **reunión de revisión del sprint:** es una reunión, donde se vuelven a reunir todos los interesados (cliente, Product Owner, líder del equipo y el resto de miembros). En ella se revisa el producto que se ha creado, normalmente mediante una demo, y el cliente lo valora. Si alguna característica no satisface al cliente, vuelve al Product Backlog (Pila del Producto) y se vuelve a redefinir su prioridad.

Dentro de los tipos de reuniones descritas anteriormente, se pueden identificar más subtipos. Estos subtipos consisten en hacer reuniones aparte para tratar asuntos muy concretos que conciernen a un tipo de trabajadores. Por ejemplo, si hay un problema técnico de infraestructura en el proyecto, para esa reunión sólo participarían los miembros del equipo que sean administradores de sistemas.

d) La Mecánica de Scrum

Después de mencionar los elementos fundamentales en los que se basa Scrum en este apartado se muestra cómo aplicar el método correctamente, relacionando así, artefactos, reuniones, roles y sprints.

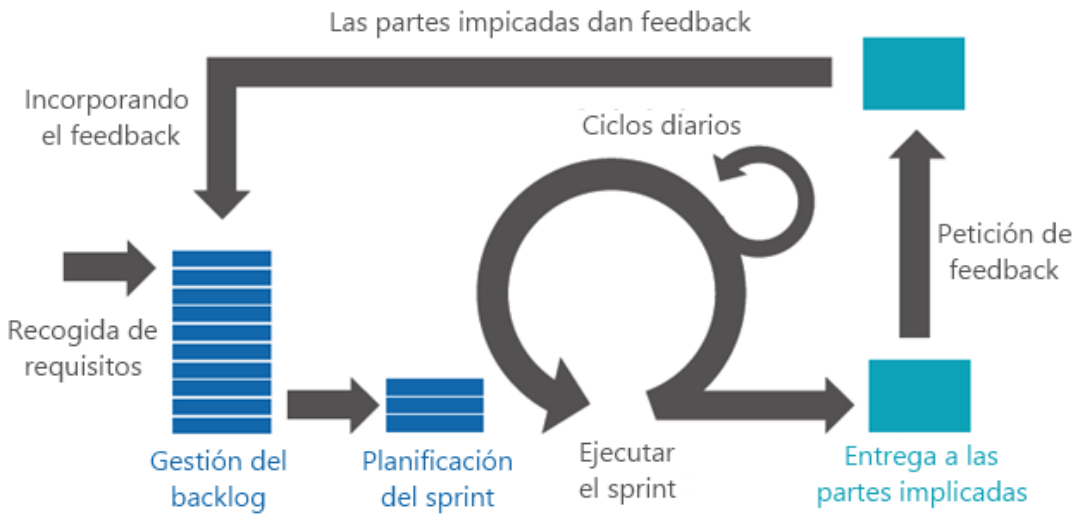

Ilustración 6: Actividades de Scrum (80)

En la ilustración 6 se representan las actividades y artefactos que forman parte del modelo Scrum, y que se explican a continuación.

El desarrollo empieza cuando el propietario del producto (Product Owner) tiene la visión del producto. De esta visión se deducen características que serán catalogadas y clasificadas en función de, si suponen riesgos para el proyecto o son valor de negocio. Estas características se convertirán en los requisitos funcionales. Así se obtiene el primer artefacto, el Product Backlog o Pila del Producto.

La pila del producto está en una continua evolución y, por lo tanto, no va a permanecer estática durante el desarrollo del sistema software. En función de la capacidad de trabajo del grupo, se seleccionarán y estimarán las tareas más importantes y, a la vez, más rápidas de crear, para que el cliente puede ver resultados del primer sprint en poco tiempo. A este proceso se le conoce como Planificación del Sprint. En este momento el Product Owner ha acabado su trabajo y le traslada al equipo de desarrollo las tareas que tendrán que realizar (Sprint Backlog).

El equipo de desarrollo realiza labores de codificación, pruebas y despliegue, entre otras tareas técnicas. También deben hacer una estimación del tiempo que tardarán en realizar estas tareas. Las reuniones diarias (Daily Meeting) sirven para sincronizar el trabajo del grupo o para hacer pequeños feedback a ellos mismos. Cuando el equipo termine su tarea, se obtiene un artefacto que entregar y mostrar al cliente. Sin embargo, no es suficiente con que pase las pruebas unitarias, hay que entregar un artefacto

potencialmente explotable, es decir, una demostración funcional de los progresos realizados en el sprint.

Cuando este proceso acaba, se lleva a cabo una reunión con el cliente (Sprint Review Meeting) y con todos los implicados, para hacerles una demostración de lo realizado en el último sprint. En ella los clientes darán el visto bueno o no, a la tarea realizada. Acabada esta reunión, el equipo tiene una última reunión para discutir sobre el proceso (Scrum Restrospective), no el producto. Es decir, se comprueba si se están haciendo las cosas bien, en qué se ha fallado etc. Después de esta última reunión se vuelve a repetir todo el proceso, comenzando por el principio la pila del producto.

El producto final será una sucesión de sprints. El proceso de desarrollo acaba cuando ya no quedan más requisitos o funcionalidades que implementar.

3.2.2. XP (Extreme Programming)

La creación de XP se remonta a 1996 cuando un ingeniero de software llamado Kent Beck comenzó a trabajar para el fabricante de automóviles Chrysler (81). Recién llegado a la empresa tuvo el atrevimiento de plantear a sus superiores que el proyecto al que estaba asignado era un desastre y que había que desechar todo el código fuente implementado hasta ese momento, y comenzar de nuevo. Sus jefes aceptaron la propuesta y lo dejaron a cargo del proyecto. Fue en este momento cuando Kent pudo aplicar sus ideas y poner en marcha de manera experimental “su metodología”, que en ese momento era una versión muy inicial de Extreme Programming.

Pasado unos años, Kent presentó su libro “Extreme Programming Explained” donde explicaba de qué manera estaba gestionando el proyecto en el que estaba involucrado. En él detalla la metodología Extreme Programming de manera completa. Esta fue la primera versión de XP (1999) que hoy día no se usa, puesto que poco después, el proyecto fracasó y fue cancelado en el año 2000 porque, según Chrysler, el presupuesto fue muy superior al asignado inicialmente.

Este fracaso sirvió para aprender y perfeccionar la metodología Extreme Programming y sacar adelante su segunda versión con nuevos valores en año 2004. Esta segunda versión es la que se usa hoy en día y, por lo tanto, la que se sigue aplicando.

Siguiendo escrupulosamente el método descrito por Kent en sus libros, XP se compone de cuatro pilares fundamentes que son: los valores, los principios, las prácticas y los

roles. En función de si tratamos con la primera versión o la segunda, cambian estos pilares ya que la segunda versión fue un rediseño radical de la metodología.

a) Los Valores

Para que un grupo sea efectivo usando XP, cada uno de sus miembros debe de tener estos cinco valores:

- **Comunicación**: Dentro de un equipo de desarrollo puede ocurrir que haya falta de comunicación entre los programadores. Esto repercute en el proyecto negativamente, puesto que se pueden dar situaciones como: que un programador tome una decisión por su cuenta que afecta al proyecto de manera crítica, o que un miembro del equipo oculte retrasos del proyecto para evitar una reprimenda. Para XP la comunicación del grupo es esencial, puesto que, al ser una metodología ágil, el factor humano es determinante para avanzar en el desarrollo del software. El flujo de información, por lo tanto, es vital y debe existir una buena comunicación entre todos los miembros del equipo.
- **Simplicidad**: Para un proyecto ágil la simplicidad del código es muy importante, así como la arquitectura del software. Cuanto más sencilla sean ambas, mejor para el desarrollo del proyecto.
- **Retroalimentación (Feedback):** El feedback es un valor que necesita de comunicación y simplicidad para que pueda resultar efectivo. En un grupo de desarrollo se debe dar varios tipos de feedback: diarios, semanales o mensuales. Esto lleva, por un lado, a que el equipo se encuentre en un contacto continuo y directo entre ellos, y por otro, a un flujo de comunicación y validación ininterrumpido con el cliente.
- **Valor**: El equipo no debe tener miedo a enfrentarse al desarrollo y a los futuros problemas que surgirán. El equipo tiene herramientas y métodos para combatir los problemas, y junto con los valores anteriores pueden resolver la situación. En situaciones como: desechar código que no sirve, revertir cambios de requisitos ya creados o posibles retrasos, el equipo de desarrollo debe mostrar valor y coraje para sacar el proyecto adelante y hacer los cambios o modificaciones necesarias sin miedo de fracasar.
- **Respeto**: Este valor fue añadido en la segunda versión del 2004 de XP. El respeto hacia los compañeros es fundamental, y debe convivir con los valores

anteriores. XP es un método que utiliza en gran medida las relaciones humanas y se debe tener respeto y ser tolerante con la actuación de los demás compañeros, puesto que, si no se tuviera respeto, podría ponerse en peligro el desarrollo del software.

b) Los Roles en XP

A diferencia de Scrum, XP tiene por base más roles con tareas específicas para cada persona.

- **Programador**: Es la base del equipo. Su labor consiste en implementar el código fuente del software y las pruebas o test que hay que lanzar.
- **Cliente**: El encargado del rol del cliente. Debe deducir y priorizar las historias de usuario que se crean y se implementan en cada una de las iteraciones, en función del valor de negocio que proporcionen al producto software. Además de las historias de usuario también escribe las pruebas funcionales que implementara los programadores.
- **Tester**: Las personas asignadas con este rol ayudan al cliente a escribir las pruebas funcionales y recolecta la información de los resultados y la difunde en el equipo. En este rol es muy importante el respeto, el tester no debe humillar al equipo de programadores, está para informar de los errores que haya en el código.
- **Supervisor** (Tracker): Es el encargado de proporcionar feedback al equipo. Su labor es continua, supervisa el estado del desarrollo y comprueba que el proyecto va en la dirección correcta y que progresa adecuadamente. Además de supervisar, tiene que estimar y planificar el tiempo, presupuesto y otros factores determinantes para el proyecto software.
- **Entrenador (**Coach**)**: El coach debe guiar al equipo durante toda la duración del proyecto. El entrenador es una persona con amplios conocimientos sobre XP que vigila que se cumplen las 12 prácticas de esta metodología.
- **Consultor**: El consultor es un miembro que no forma parte en el equipo, será una persona externa que resuelve dudas sobre factores concretos del proyecto: requisitos, lenguaje del cliente etc.

- **Jefe del Proyecto (Big Boss):** Comunica las decisiones entre el cliente y el proyecto y coordina labores del equipo desarrollo del proyecto. Este rol puede asociarse con el de supervisor, de forma que sean la misma persona.

c) Los Principios de 1999

Los valores que hemos visto anteriormente se materializan por medio de una serie de principios, concretamente 15, que se encuentran recogidos en el libro de Kent. Estos 15 principios derivan de cinco fundamentales que son los que se van a comentar a continuación. Esta serie de principios son intermediarios entre los valores y las prácticas de XP. Los diez restantes sólo se mencionarán.

- **Realimentación rápida (*Rapid feedback*):** Kent indica que debe existir una recogida de feedback diaria, semanal y mensual, de forma que vayamos validando los fallos continuamente y durante todo el desarrollo.
- **Asumir simplicidad (*Assume simplicity*):** Buscar soluciones simples para problemas complejos. Para intentar ayudar a resolver un problema el equipo debe buscar la comunicación entre los demás miembros del grupo y así encontrar una solución óptima.
- **Cambio incremental (*Incremental change*):** Cuando en una organización se introducen grandes cambios, como es cambiar su metodología a XP, el cambio debe ser progresivo y no de golpe. XP indica que el cambio debe ser de poco en poco y que se incremente a cada iteración. Se hace de esta manera para que los miembros del equipo de desarrollo se habitúen a los pequeños cambios y desarrollen una rutina de trabajo nueva, y una vez asimilada esta rutina ir poco a poco introduciendo estos cambios.
- **Abrazar el cambio (*Embracing change*):** Este principio va de la mano con el anterior. XP dice que los miembros del equipo junto con los valores de XP deben aprender a aceptar la nueva forma de trabajo sin que ello le suponga disconformidad.
- **Trabajo de calidad (*Quality work*):** El código, la documentación y los demás artefactos deben ser los óptimos para el proyecto y Kent describe que hay que perseguir el concepto de “la excelencia” como resultado del proyecto.

De estos principios derivan otros diez más que no se estudiarán, pero son básicamente profundizaciones de estos cinco:

- Enseñar a aprender (*Teach learning*)
- Pequeña inversión inicial (*Small initial investment*)
- Juega para ganar (*Play to Win*)
- Experimentos concretos (*Concrete experiments*)
- Comunicación abierta y honesta *(Open honest comunication*)
- Trabajar con los instintos de las personas, no contra ellos (*Work with people's instincts, not against them*)
- Responsabilidad (*Accepted responsibility*)
- Luz de viaje (*Travel light*)
- Medidas honestas (*Honest measurement*)

Si se consigue aplicar estos principios a un equipo de desarrollo con XP se logra alcanzar con seguridad el éxito del proyecto.

d) Los Principios de 2004

En la segunda versión de XP (81) se rediseño la estrategia a seguir, y para ello, el primer paso que dio Kent fue rediseñar los principios sobre los que se sustenta esta metodología. En esta nueva versión, los principios pasaron a ser 14, uno menos que antes, y se modificó radicalmente la base de XP. A continuación, se explica brevemente los 14 principios nuevos, que son importantes para generar las prácticas a realizar en XP.

- **Humanidad (*Humanity*):** Un equipo de XP está formado por personas, que tiene necesidades básicas de cualquier ser humano: seguridad, realización, pertenencia al grupo, crecimiento personal etc. Este principio hace referencia a equilibrar estas demandas humanas con las necesidades del equipo.
- **Economía (*Economics*):** El principio de la economía hace referencia a asegurar el valor económico del proyecto. El desarrollo tiene que satisfacer a los objetivos de la organización.

- **Beneficio Mutuo (*Mutual Benefit*):** Las prácticas que se apliquen durante el proyecto tiene que beneficiar a todos los integrantes del equipo.
- **Auto-similitud (*Self-Similarity*):** Es el uso de técnicas similares para problemas que surjan durante el desarrollo.
- **Mejora (*Improvement*):** El objetivo es mejorar en cada paso e ir refinando el proyecto poco a poco.
- **Diversidad (*Diversity*):** La solución a los problemas que surgen durante el desarrollo del proyecto no siempre tienen una única solución. Si se presentan varias soluciones posibles tiene que ser una oportunidad para el desarrollo y no un motivo de confrontación.
- **Reflexión (*Reflection*):** Los errores que se comenten durante el proyecto deben exponerse al resto del equipo para aprender y mejorar, en ningún caso se deben de ocultar. A cada paso que se de en el desarrollo se tiene que pensar cómo y porqué hay que darlo.
- **Flujo (*Flow*):** Las diferentes fases del proyecto deben suponer un flujo de información entre ellas.
- **Oportunidad (*Opportunity*):** Los problemas que se tengan en el proyecto son oportunidades y no problemas.
- **Redundancia (*Redundancy*):** Los problemas críticos, soluciones o partes del proyecto complejas, hay que revisarlos varias veces y de diferentes formas.
- **Fallo (*Failture*):** Los fallos no son en vano, siempre se puede sacar provecho de ellos. Hay que aprender de los errores.
- **Calidad (*Quality*):** Hay que dar lo mejor de uno mismo, es preferible ir más despacio para ganar calidad, que ir rápido y hacer un producto con errores.
- **Pequeños Pasos (*Baby Steps*):** Los problemas complejos se resuelven mejor dividiéndolos en pequeñas partes y atajándolos de poco en poco.
- **Responsabilidad (*Accepted responsibility*):** Las responsabilidades no pueden ser asignadas. Si una persona acepta una responsabilidad debe hacerse cargo de ella. Con la responsabilidad llega la autoridad.

e) Las Prácticas

Las prácticas de XP2 (La segunda versión de XP del 2004) se dividen en dos categorías (82), que a su vez tienen varios apartados. La división se realiza en función de la prioridad de las prácticas. Hay que ejecutar primero las prácticas primarias y una vez aplicadas, ejecutar las prácticas de corolario. En este apartado se mencionarán brevemente las prácticas (83) que usa XP, sin entrar en detalles:

1) Prácticas Primarias
    - Análisis y Planificación de Requisitos
        - Historias de Usuario
        - Ciclos semanales
        - Ciclos mensuales
        - Holguras (*Slacks*)
    - Equipo y Factores Humanos
        - Sentarse juntos (*Sit Together*)
        - Todo el equipo (*Whole Team*)
        - Espacio de trabajo informativo (*Informative Workspace*):
        - Trabajo con ganas (*Energized Work*)
        - Programación en parejas (*Pair Programming*)
    - Diseño
        - Diseño incremental (*Incremental Design*)
        - Prueba primero, programa después (*Test-First Programming*)
    - Codificación y Pruebas
        - Construcción en diez minutos (*Ten-Minute Build*)
        - Integración continua (*Continous Integration*)

2) Prácticas de corolario
    - Planificación y Análisis de Requisitos

    - Participación real del cliente (*Real Client Involvement*)
    - Desarrollo incremental (*Incremental Deployment*)
    - Negociar alcance de contratos (*Negotiated Scope Contract*)
    - Pagar por el uso (*Pay per Use*)
- Equipo y Factores Humanos
    - Continuidad del equipo (*Team Continuity*)
    - Reducción del equipo (*Shrinking teams*)
- Diseño
    - Análisis raíz-causa (*Root-Cause Analysis*)
- Codificación y Pruebas
    - Código compartido (*Shared Coded*)
    - Código y pruebas (*Code and Test*)
    - Versión Única (*Single Code Base*)
    - Implementación diaria (*Daily Deployment*)

f) El Método

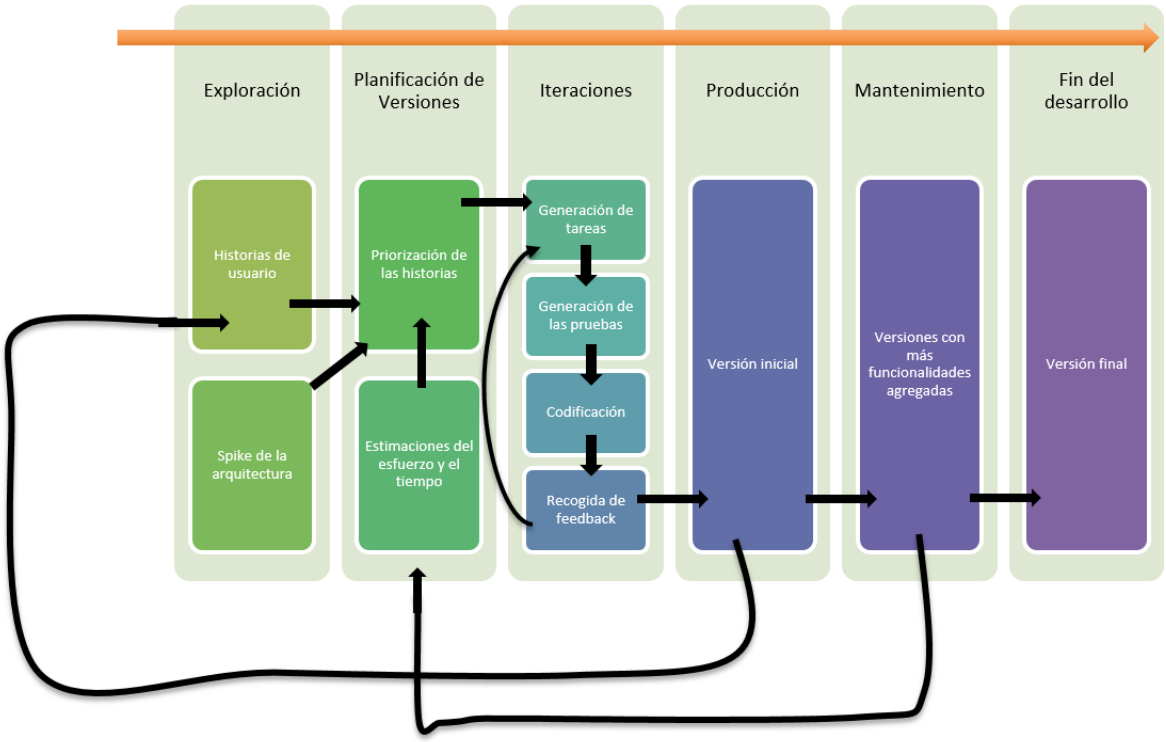

Ilustración 7: Ciclo de desarrollo de XP

La aplicación de XP consta de 6 fases diferenciadas y secuenciales tal y como muestra la ilustración 7.

- **Exploración**: En esta primera fase el cliente y los desarrolladores se ponen en contacto y utilizan la técnica de las historias de usuario para describir los casos de uso y los requisitos funcionales del software que quiere el cliente. De esta forma los desarrolladores pueden ir construyendo la arquitectura del software. También se crea en esta fase los "*spikes*" que son programas muy simples para explorar potenciales soluciones a problemas relacionados con el desarrollo (84).
- **Planificación de versiones**: Una vez obtenido los casos de uso y los requisitos funcionales, el equipo de desarrolladores inicia una selección en la que priorizan los requisitos que se tienen que satisfacer cuanto antes. Los requisitos seleccionados serán los que se implementen en la fase siguiente. También se

hace una estimación del tiempo y el esfuerzo que tendrá que soportar el equipo para realizar dicho requisito.

- **Iteraciones**: En esta fase se implementa el código de los requisitos que fueron seleccionados en la fase anterior. Para ello, se crea una lista de tareas a realizar y se asigna estas tareas a una pareja de programadores que deberán llevarla a cabo. Además de implementar el código, tiene que crear las pruebas y ejecutarlas. Después, añadirán su aportación al resto del código del proyecto y volverán a ejecutar pruebas, esta vez de regresión. Después de pasar estas pruebas, el cliente debe ejecutar pruebas de aceptación y el equipo recogerá feedback, para iniciar un nuevo proceso de iteración.
- **Producción y mantenimiento**: Estas dos fases, engloban la recogida de artefactos que el cliente ha aprobado en la fase previa. El código que ha llegado a estas dos fases es un código muy verificado y no debería presentar problemas. Los artefactos recogidos aquí están listos para ser sometidos a producción. Si en iteraciones futuras se añade código, serán nuevas funcionalidades. El mantenimiento es una fase imprescindible para compactar todo el código del proyecto.
- **Final del desarrollo**: El final del desarrollo se produce cuando se han acabado por implementar todas las funcionalidades que el cliente demandaba, es entonces cuando se produce la documentación externa y final del proyecto, y el producto por completo pasa a ser lanzado.

3.3. CONCLUSIÓN DEL CAPÍTULO

En este capítulo se han revisado los conceptos básicos de las metodologías ágiles, y además, se han expuesto las características principales de las metodologías ágiles más usadas y conocidas: SCRUM y XP.

Una parte fundamental del desarrollo ágil es la obtención de software de calidad. En las páginas anteriores se ha comentado cómo se logra dicha calidad a través de la puesta en práctica de valores como: valorar más a las personas frente al código, hacer hincapié en las pruebas, no crear una documentación excesiva etc. Aun así, en muchos casos sigue siendo insuficiente. En el siguiente capítulo se detalla una nueva tendencia para probar software, que promete mayor calidad frente a las tradicionales fases de

pruebas. En este nuevo tipo de pruebas evitar errores es el objetivo primordial. Poniendo Scrum como referente se estudia el *agile testing* en el siguiente capítulo: Pruebas en el mundo ágil.

4. PRUEBAS EN EL MUNDO ÁGIL

En capítulos anteriores se han mostrado una visión general de los desarrollos tradicionales, las características generales de las pruebas, y los criterios de calidad del software. En este capítulo se estudian las características y particularidades de las pruebas y las prácticas para metodologías ágiles, más famosas, reconocidas y usadas en la industria del software.

Tomando Scrum como referencia, hay que analizar dónde o en qué momento se realizan las labores de testing en un desarrollo ágil. En el capítulo anterior se ha comentado que al igual que, en las metodologías tradicionales, en Scrum las pruebas se realizan al final de cada sprint. Parece que las labores de prueba y testeo en este caso, también se realizan al final. Es cierto que en Scrum se realiza bastantes más tareas de testing por el simple hecho de que hay que realizar labores de prueba cada vez que finaliza un sprint y de esta forma es más fácil llegar a la calidad que el usuario desea. En este sentido Scrum cumple un aspecto de calidad esencial en el software.

Según la filosofía Scrum, un proyecto tiene muchas entregas parciales, lo que supone probar el producto muchas veces. Cuando un sprint está en marcha es como un mini proyecto, ya que consta de todas las fases de un desarrollo, divididas en las fases de implementación, pruebas y despliegue. Recordemos que al final de un sprint, se entrega un artefacto funcional (no se da al cliente el código fuente).

4.1. PRINCIPIOS DEL AGILE TESTING

En los últimos años ha surgido una nueva tendencia en las pruebas de software. Al igual que surgió una filosofía ágil para el desarrollo de software ¿por qué no tener una filosofía igual para las pruebas? Así aparece el nuevo término asociado al mundo del software, el agile testing o Pruebas Ágiles.

El agile testing es una práctica que se incluye en el conjunto de las pruebas de software, que sigue los principios del desarrollo ágil, involucrando en el proceso de las pruebas no solamente al equipo de desarrolladores, sino también poniendo en contacto, las pruebas con los líderes del proyecto y con el propio cliente.

El agile testing no es una fase separada del desarrollo, sino que está integrada en todas las fases, de forma que es una actividad obligatoria y tan importante como, por ejemplo, implementar el código. No comienza “después de” sino “junto con”. Esto

quiere decir que no se empiezan a hacer las pruebas cuando está el código ya implementado y finalizado, sino que se ejecutan las pruebas desde el principio.

En este paradigma, la actividad de pruebas comienza con una serie de actividades relacionadas con el análisis, la reedición e identificación de escenarios críticos para las futuras características del software (actividades dentro de la Pila del Producto). Estas actividades, reducen el tiempo de recogida de feedback, ya que se producen a lo largo de cada iteración y no al final, como se realiza tradicionalmente.

Este conjunto de actividades lleva al proyecto a una retroalimentación continua de pruebas, permitiendo corregir los fallos continuamente durante el desarrollo. Esto es muy importante, ya que en Scrum los requisitos funcionales van mutando a lo largo del proyecto.

La filosofía ágil requiere que todo el equipo se involucre en el objetivo final. Si aplicamos esta norma a las pruebas, obtenemos un cambio en las competencias de los roles. Los test no solucionan solamente los fallos, ahora representan expectativas de calidad. Además, no sólo los testers están designados para ejecutar pruebas, el agile testing involucra también a los desarrolladores.

En las pruebas ágiles es común el uso de listas de comprobación (check lists) en lugar de documentación convencional. Se define la prueba y se lleva a cabo, anotamos los resultados y se pasa a la siguiente prueba.

El principio del manifiesto ágil de no generar una documentación enorme y compleja se aplica también al testing, por el hecho de aplicar prácticas propias de agile testing, como son el uso de estilos de documentación y herramientas agiles, utilización de test automáticos o aprovechar documentos para varios propósitos.

El hecho de aplicar estos principios crea una serie características implícitas que se manifiestan por el resultado de poner en marcha estas buenas prácticas. Uno de los resultados observables de llevar a cabo agile testing es la limpieza del código. Se prueba y se recibe feedback continuamente, por lo que se mantiene un código pulcro y legible.

El código del que es responsable un desarrollador no es propio, ahora son más personas las que supervisen este código, lo que conlleva a que sea legible y limpio desde los comienzos del proyecto.

Otra característica resultante de aplicar agile testing se manifiesta en los tiempos de obtención de feedback. Obtener feedback, en este caso, es comprobar el comportamiento del código. En otros métodos de prueba este estudio se realiza al final de cada sprint, pero en el agile testing las pruebas son continuas, lo que supone no tener que esperar semanas a que el código esté completado. Lo que ocurre es que el tiempo en obtener feedback se agiliza muchísimo, es menor que antes.

4.2. PRÁCTICAS DEL AGILE TESTING

En la sección anterior, se mostraron algunos de los principios básicos que hay que considerar para realizar el *agile testing*. Pero estos principios son objetivos implícitos y explícitos que hay que llevar a cabo correctamente. Para cumplir estas normas debemos materializar los principios del agile testing, y esto se traduce en crear una serie de prácticas que se aplican en el desarrollo del proyecto.

En los apartados siguientes, se muestran las prácticas más comunes usadas en agile testing. Naturalmente existen más técnicas y prácticas para realizar agile testing, que las expuestas en este TFG, pero debido a su escaso uso, y el poco soporte por herramientas para su aplicación, se ha decidido descartarlas e incluir las más robustas y soportadas por la industria del software. Todas las técnicas excluidas van a dejar de ser soportadas a corto plazo.

4.2.1.El cuadrante de Pruebas Ágiles

En el capítulo 3 se ha visto que existe una gran variedad de tipos de prueba. Para probar el software, es necesario tener una jerarquía que organice qué pruebas usar en qué momento del desarrollo. Existe una práctica de pruebas ágil que organiza el orden de ejecución de las pruebas y detalla también cuándo hay que ser pruebas manuales y cuándo tienen que ser automatizadas (85). En la ilustración 8 podemos ver dicha matriz.

Para interpretarla debemos comenzar en el cuadrante número 1 situado en la parte inferior a la izquierda “Pruebas unitarias”. El recorrido de la matriz se realiza en el sentido de las agujas del reloj. Por lo tanto, al terminar las pruebas unitarias se pasaría al cuadrante número dos “Pruebas funcionales”, seguidamente al 3 “Pruebas de aceptación” y finalmente se acabaría en el 4 “Pruebas de carga o de estrés”.

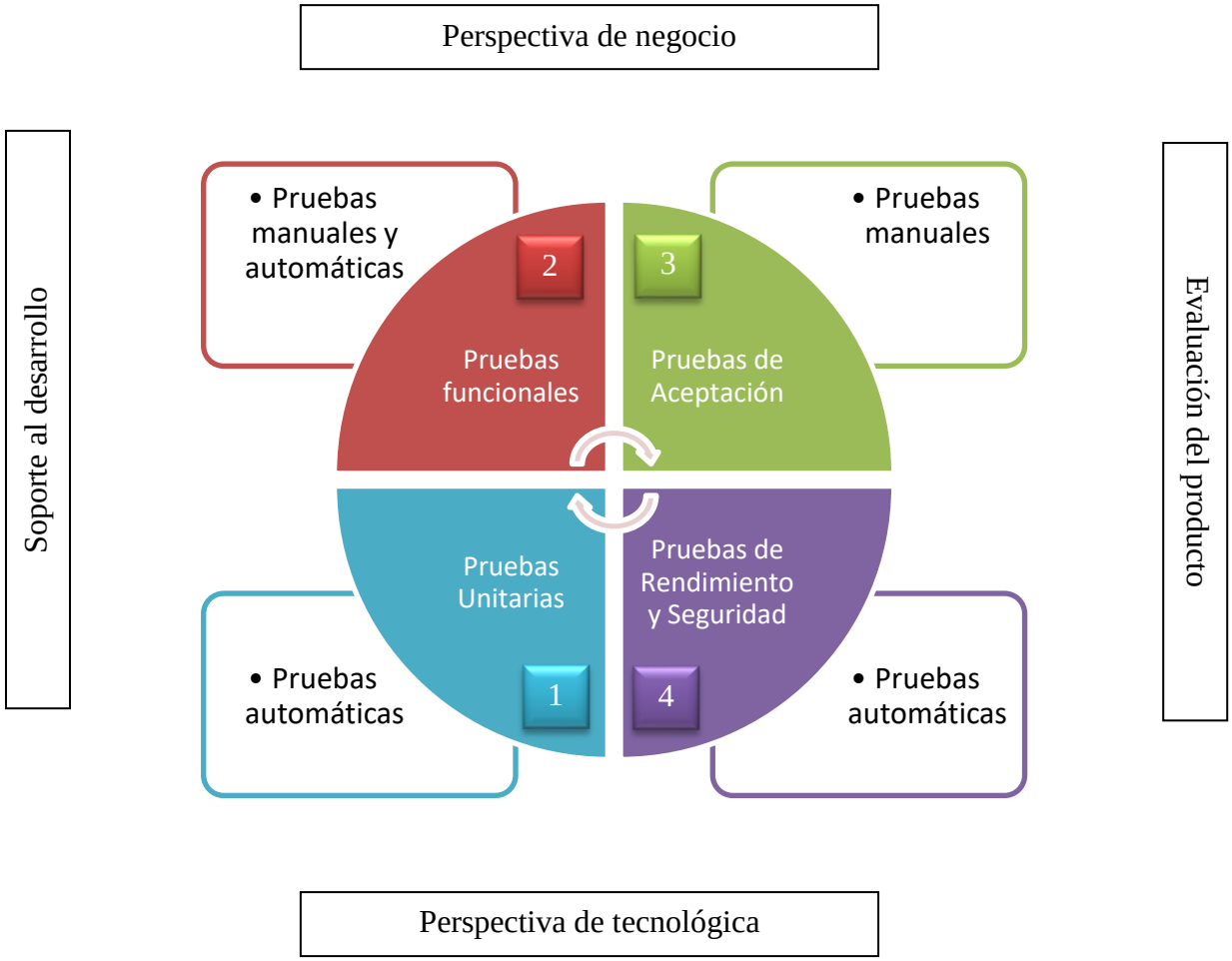

Ilustración 8: Matriz de Pruebas Ágiles

En el primer cuadrante se ejecutan las pruebas unitarias sobre el código del sistema. Se comienza por aquí porque este tipo de pruebas, son las más básicas que se pueden ejecutar sobre un proyecto y su uso es indispensable para el correcto funcionamiento de un sistema. Estas pruebas están completamente relacionadas con la perspectiva tecnológica, es decir, para la ejecución de estas pruebas se tiene que tener en cuenta la tecnología que se está usando para desarrollar (lenguaje de programación, IDE etc.). Para optimizar el tiempo invertido en la ejecución de estas pruebas, se deben automatizar la ejecución de estos test.

Después de pasar esta primera fase, se realizan las pruebas funcionales que pueden ser automáticas, manuales o ambas al mismo tiempo. Ahora se probarán aspectos relacionados con el desarrollo del proyecto en sí mismo, pero desde un punto de vista de lógica de negocio. Es decir, probar si un software realiza las operaciones que el cliente espera que haga.

Acabada esta fase se pasa a ejecutar las pruebas de aceptación. Aquí el equipo recibe *feedback* del cliente, y deberán revisar todo lo que no haya salido bien. Este tipo de pruebas sólo puede ser ejecutadas manualmente, ya que se prueba el software desde un punto de vista de usuario final. Estás pruebas evalúan el valor de negocio del software.

En la última fase se vuelve a evaluar desde un punto de vista tecnológico. Estas pruebas se realizan al final del proceso de desarrollo, y consisten en someter al producto a estrés para ver si reacciona correctamente ante situaciones extremas (pruebas de carga, de estrés, de estabilidad etc.). Aquí puede haber también pruebas relacionadas con la seguridad del mismo.

Se recomienda llevar a cabo esta técnica y en este orden para garantizar que un producto haya pasado por una batería de test mínimamente completa. Pasando esto, se garantiza que un producto funcione correctamente, al menos, en el "caso feliz".

Esta es una de las técnicas ágiles más usadas hoy en día. Obviamente, se puede complementar con más test, por ejemplo, complementar los test unitarios con test de regresión. Este cuadrante sólo índico lo mínimo que se debería hacer para probar un producto de forma ágil y a partir de esta base, continuar añadiendo más supervisión. Un buen acompañamiento a esta técnica sería usar TDD que lo veremos en el apartado 4.2.4.

4.2.2.Testing colaborativo

El aislamiento de un trabajador en un desarrollo ágil acaba resultando en un incremento del esfuerzo tanto individual como colectivo. Cuando se integran las pruebas y desarrollo se produce un código mucho más sólido. Hay sistemas que exigen una ejecución de pruebas independientes, como los sistemas críticos, pero eso no implica que los únicos que estén probando el código sean los testers. El testing colaborativo es un esfuerzo constante, ya que en agile testing las pruebas no son una fase más, sino que se producen a lo largo de todo el desarrollo del proyecto, que depende de todo el equipo de desarrollo; ya no hay testers todo el equipo se encuentra involucrado en las pruebas.

4.2.4.Test Driven Development (TDD)

El Test Driven Development o Desarrollo Dirigido por Test, en adelante TDD, es una técnica de diseño e implementación de software procedente de la metodología XP. El

nombre que recibe esta técnica de diseño es algo ambiguo, ya que sugiere que se utiliza para crear muchos test sobre el código, pero no es así. TDD se usa para diseñar e implementar correctamente los requisitos del software. Es una técnica que resuelve las preguntas que todo programador tiene al comienzo del desarrollo del software. Responde a cuestiones tan importantes como por dónde empezar, como hacerlo, o qué debemos codificar y que no.

En Scrum, el rol de Scrum Master es el encargado de decidir qué implementar en cada sprint del desarrollo y no los desarrolladores. Pero, por otra parte, aplicando TDD se responde a cuestiones referentes a la implementación, y a la vez, a la toma de decisiones, competencias que son mutuamente excluyentes en Scrum. Esto es debido a que TDD ayuda tanto al desarrollador, a codificar su código, como al arquitecto del software con la toma de decisiones, puesto que los test generados al usar esta técnica de diseño desvelan el dominio del negocio y el diseño del software.

Para usar TDD el desarrollador de software tiene que cambiar su método de trabajo. Ahora, en vez de crear los casos de uso y después implementarlos, va a intentar convertir un caso de uso en un ejemplo. Si se realiza esto con todos los casos de uso, llegará un momento en el que los ejemplos describirán por sí solos la próxima tarea a implementar.

Con esta forma de proceder, se eliminan las ambigüedades que antes podían surgir por usar el lenguaje natural, puesto que con el cliente no se habla formalmente. Los ejemplos van a mostrar qué hay que implementar en el sprint y también van a desvelar la arquitectura que se debe usar: si es una aplicación para un ordenador personal, para un teléfono, etc. Es evidente que, si el proyecto es de una rama muy concreta, no se va a necesitar ni si quiera los ejemplos para saber que se tendrá que usar servicios web, por ejemplo.

El objetivo de esta técnica es minimizar la creación del código implementando sólo lo imprescindible, lo que el usuario ha pedido, reduciendo los errores y creando software con gran aceptación ante cambios (siguiendo la filosofía ágil).

La manera de proceder con TDD se basa en tres fases: crear los ejemplos (que será la especificación del caso de uso), implementar el código para realizar el test satisfactoriamente y refactorizar el dicho código (Ilustración 8).

Ilustración 9: Fases del Diseño Dirigido por Test

En la primera fase se **crea la especificación del requisito**, el ejemplo o el test. La primera duda es cómo se crea un test sin tener el código implementado. La clave reside en tener la capacidad de imaginar la forma del código que se implementará y cómo testear que funciona correctamente. En un principio no se tienen test completos, si no que se tienen ejemplos o especificaciones. Cada una de estas especificaciones individualmente pasará por las demás fases del TDD. Esto se hace así, porque diseñar la especificación y las pruebas antes de llegar a implementarla hace que el código sea radicalmente diferente.

Una vez completada la especificación del requisito o el test, se pasa a **implementar el código del ejemplo** que lo hará funcionar. Sólo hay que implementar lo imprescindible para realizar la prueba, sin importar la calidad o limpieza del código. Por lo tanto, el objetivo en esta etapa es implementar la funcionalidad para que el test se ejecute correctamente. En esta fase pueden surgir dudas al desarrollador sobre cómo interactuará el futuro flujo de información de la aplicación: bucles, condiciones, entrada y salida de datos etc. La manera de seguir es anotando estas dudas, que serán usadas en sprint futuros.

En la última fase, **se refactoriza el código**. Esto quiere decir que el programador buscará en el código, líneas y funciones duplicadas o errores lógicos. Hay que revisar no solamente el código de la prueba, también el del ejemplo.

Para llevar a cabo esta técnica se necesitan aplicar herramientas de software que ayuden al programador.

Las herramientas de código abierto más conocidas para aplicar TDD son:

- Karma (15)
- Maven (50)
- SpecFlow (86)
- Gradle TestKit (87)

Entre las herramientas comerciales más conocidas destacan:

- EggPlant (88)
- ThoughtWorks Twist (89)

4.2.5.Behaviour Driven Development (BDD)

El Desarrollo Basado en Comportamiento o BDD, surge de la mano del ingeniero Dan North en el año 2006 (90). Muchos autores apuntan que debería llamarse Desarrollo Guiado por Funcionalidad, puesto que esta técnica está basada en el desarrollo software por funcionalidades o características.

BDD está basado en la técnica TDD, y corrige ciertas deficiencias que presenta TDD. El principal problema de TDD se produce cuando un software pasa correctamente un test y no consigue la funcionalidad deseada. Que un software pase correctamente un test no significa que haga lo que el cliente esperaba de él.

La diferencia principal con TDD, es que BDD trabaja con historias de usuario en lenguaje natural, un lenguaje que tanto programadores como clientes entienden. Y, además, trasforma ese lenguaje formal a "lenguaje máquina", es decir, código que está dividido en tantos módulos como características hayamos descrito en las historias de usuario.

El objetivo de BDD es que las historias de usuario dirijan el desarrollo del proyecto software. Además, BDD permite comprobar que el software implementado cumple con la funcionalidad que se esperaba, corrigiendo así la deficiencia de TDD.

a) Gherkin

En BDD, Gherkin (91) es el término utilizado para denominar el lenguaje compartido entre: los programadores y el cliente, con el ordenador. Los desarrolladores junto con el cliente tienen que escribir historias de usuario, para facilitar a los desarrolladores la extracción de los casos de uso del software que quiere el cliente. Gherkin no sólo se utiliza para describir las funcionalidades en lenguaje natural y código, también se

utiliza para crear pruebas automatizadas, en la mayoría de los casos pruebas unitarias[10], y para generar documentación interna.

Las historias de usuario (User Story) es una técnica que facilita la extracción de características o funcionalidades que debe tener el software. Gherkin utiliza esta herramienta y la trasforma en un lenguaje compartido entre humanos y máquinas para desarrollar el software.

Para escribir de las historias de usuario con Gherkin, lo primero que hay que conocer es la sintaxis que utiliza, que se basa en cinco palabras que crean sentencias con las que se va formando la historia.

- **Características (Features):** Las características son frases, en las que el cliente y los desarrolladores se ponen de acuerdo, y especifican cómo debe actuar el software ante un evento generado por el usuario.
- **Escenario (Scenario):** Las características o funcionalidades se definen mediante escenarios, que representan situaciones en las que se pueda encontrar el usuario frente al software en algún momento.
- **Precondiciones (Given):** Para cada escenario generado, se crea una serie de pasos que el usuario debe seguir hasta llegar al final del escenario. Esta serie de pasos tienen precondiciones que el usuario y el software deben cumplir para continuar: o bien al siguiente paso o bien a la siguiente precondición.
- **Acciones (When):** Cuando el usuario cumple las precondiciones, se produce una serie de interacciones o acciones con el software, que se registran en la historia de usuario. El software ante estas acciones debe responder con un resultado.
- **Resultados (Then):** Son la respuesta del software a las acciones que previamente tomó el usuario.

Para crear una característica (feature) se utiliza una frase a modo de ayuda (92) que va cambiando en función de la característica que se desee implementar:

[10] Pruebas con Gherkin: Existen herramientas que automatizan pruebas que no son unitarias, por ejemplo, de entorno, pero están fuera del lenguaje Gherkin.

As a [role] I want [feature/goal/desire] so that [benefit]

Yo como [personaje/ rol] quiero [característica] para obtener [beneficio]

Definida las características o las funcionalidades, se plantean escenarios posibles que puedan ocurrir con la característica en cuestión, y a partir de aquí se genera pasos para que la característica acabe con éxito. Los pasos constan de precondiciones y acciones que llevan a un resultado.
Un ejemplo muy sencillo es el siguiente: primero definimos la característica o funcionalidad a implementar en la historia de usuario.

> ***Característica****: un usuario quiere acceder a una plataforma web universitaria para descargar sus apuntes.*

A partir de esta historia se desarrollan varios escenarios, que en el fondo serán casos de uso:

> ***Escenario 1****: El usuario entra correctamente*
> ***Tiene*** *un nombre de usuario y una contraseña*
> ***Y*** *el nombre de usuario existe*
> ***Y*** *la contraseña es correcta*
> ***Entonces*** *el usuario puede acceder a su cuenta y su información*
> ***Y*** *puede descargar sus apuntes*
>
> ***Escenario 2:*** *El usuario no recuerda la contraseña*
> ***Tiene*** *un nombre de usuario*
> ***Cuando*** *el usuario hace clic en "No recuerdo la contraseña"*
> ***Y*** *el usuario inserta su nombre de usuario*
> ***Y*** *el nombre de usuario existe*
> ***Entonces*** *se envía un correo de verificación*
> ***Y*** *el usuario debe validar el enlace*
> ***Y*** *el usuario debe insertar nueva contraseña*
> ***Entonces*** *el usuario tiene que volver a entrar con su nueva contraseña*
> ***Y*** *se ejecuta Escenario 1*

Una vez definida, BDD utiliza herramientas para convertir esta historia de usuario en código. Para ello se utilizan "contenedores" o frameworks; hay tantos contenedores como lenguajes de programación haya. Normalmente están disponible para la mayoría de los lenguajes de programación más usados:

Tabla 2: Frameworks para los lenguajes

Lenguaje	Framework
Java	Concordion, JDave y Easyb
Javascript	Jasmine
Ruby	Cucumber
Phyton	BeHave
PHP	Behat y Kanlan

La conversión del lenguaje natural a código realizada con estos frameworks puede parecer ambigua. Siguiendo el ejemplo anterior, un posible resultado de aplicar un framework al ejemplo en Gherkin es el siguiente:

```
$Jasmine.
Feature: El usuario entra correctamente

Scenario1: El usuario entra correctamente #feaures/campusvirtual.feature:1
Given un nombre de usuario y una contraseña
And el nombre de usuario existe
And la contraseña es correcta
Then el usuario puede acceder a su cuenta y su información
And puede descargar sus apuntes


1 Scenario (1 undefined)
5 Steps (5 undefined)
You can implement undefined steps with these code snippets:

/**
 * @Given /^el nombre de usuario y una contraseña (\name + password)$/
 *
 */
public function comprobarUsuario($user, $password)
{

  throw new CheckException();

  }
```

Ilustración 10: Ejemplo con Jasmine

Como se puede ver en la ilustración 9 la conversión ha sido sencilla. Por cada característica definida se crea una función, que implementarán los programadores, y de esta forma se crea el esqueleto de la aplicación software. Dentro de cada característica, se crea también una función por cada precondición (given), acción (when) y resultado (then).

Cuando aplicamos este método en varios sprint, se obtiene software muy estructurado y ordenado. Aunque BDD sea una técnica muy reciente está teniendo una acogida muy buena por parte de los desarrolladores y cada vez es más relevante.
Existe una amplia variedad de herramientas y frameworks para aplicar BDD. Ya se han presentado los frameworks por lenguajes que soportan esta técnica. Además de los ya presentados, otras herramientas conocidas son:

- SpecFlow (86)
- Calabash (93)
- Robot Framework (23)
- Codeception (94)
- Squish (95)
- Frank (96)

Las herramientas comerciales más conocidas destacan:

- EggPlant (88)
- ThoughtWorks Twist (89)

4.2.6.Acceptance Test Driven Development (ATDD)

El Desarrollo Dirigido por Test de Aceptación o Acceptance Test Driven Development, es otra técnica de diseño muy similar al TDD ya comentado. Hay autores que se refieren a esta técnica como Story Test Driven Development (STDD), que recibe este nombre debido al comportamiento de la misma.

La idea es la siguiente: se parte de un desarrollo al que los programadores han aplicado TDD. Si lo han hecho correctamente, deberían haber obtenido un código limpio y refactorizado, que ha sido probado por pruebas unitarias. El código, por lo tanto, es bueno, pero ¿sucede lo mismo con la aplicación?

ATDD no es una técnica de desarrollo o una buena práctica; tiene un enfoque superior. Es una metodología de trabajo para desarrollar software, que busca comprobar si los pasos que se siguen en el desarrollo son los mejores para llegar al objetivo. Con TDD sólo se ve si lo que se desarrolla está codificado correctamente y sin errores, es un

enfoque miope, sólo centrado en desarrollar código de calidad, no la aplicación en su conjunto.

Al igual que ocurre en TDD, ATDD consta de una serie de pasos secuenciales, pero en ATDD no basta con crear ejemplos y sus test. En esta técnica el jefe del equipo, la voz del cliente y los tester tienen que crear una historia de usuario (User Story) y con ella deberán proponer, crear y aceptar unos criterios de aceptación, con ejemplos, para luego ejecutar las pruebas de aceptación, creadas en base a los criterios propuestos, antes de comenzar con el desarrollo del proyecto. Las historias de usuario es una herramienta para comprender mejor la idea del cliente. Es parecido a los casos de uso, sólo que, en vez de tener texto o diagramas por escrito, se tiene una idea en el argot del cliente, que permitirá definir mejor lo que el cliente pide. Usando ATDD se crean pruebas de aceptación antes de desarrollar cualquier requisito del software.

En la ilustración 10 se puede ver con mayor claridad.

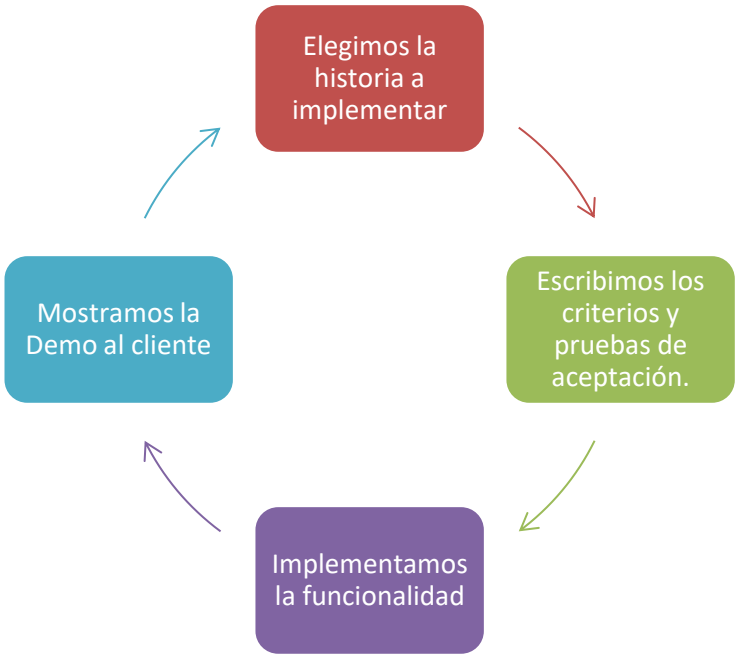

Ilustración 11: Fases del Desarrollo Dirigido por Test de Aceptación

Esta técnica comienza con una reunión entre los miembros del equipo. Es necesario que esté presente el equipo de desarrollo y el cliente, o en su defecto la voz del cliente. Si se estuviera usando una metodología Scrum, esta primera reunión se celebraría al inicio del Sprint, antes de que el Scrum Master preparase la Scrum Backlog, y a ella asistiría el Scrum Master, Product Owner, Tester y los desarrolladores del equipo. En

esta reunión inicial, se aborda qué funcionalidad desarrollar primero y con ella qué **historia de usuario** se creará en el sprint. También se ponen de acuerdo en tener unos criterios y pruebas de aceptación comunes. Se responde así, a preguntas básicas del desarrollo del producto.

En fases sucesivas del desarrollo, al celebrar esta reunión se va confirmando que la experiencia de usuario es satisfactoria, que los requisitos planteados y desarrollados son necesarios y que el equipo ha tenido en cuenta más escenarios posibles.

Una vez finalizada esta reunión se **escriben las pruebas de aceptación**, en lenguaje natural. En de esta fase se crean los ejemplos en los que se tiene que identificar las variables y los valores de las pruebas. Si el cliente usara un vocabulario de palabras pertenecientes al sector o próximas al cliente, se usa también este argot para describir las pruebas de aceptación. En etapas intermedias del proyecto, esta fase sirve para refinar las pruebas de aceptación o crear nuevas.

La siguiente fase está dividida en dos etapas. En primer lugar, se **implementa el requisito** (ilustración 11), la prueba, automatizando los ejemplos creados anteriormente, que se quieren transformar en pruebas. Después se tendrá que **aplicar TDD** (ilustración 12) al código implementado. Aplicar TDD aquí es necesario para obtener un código limpio y sin errores.

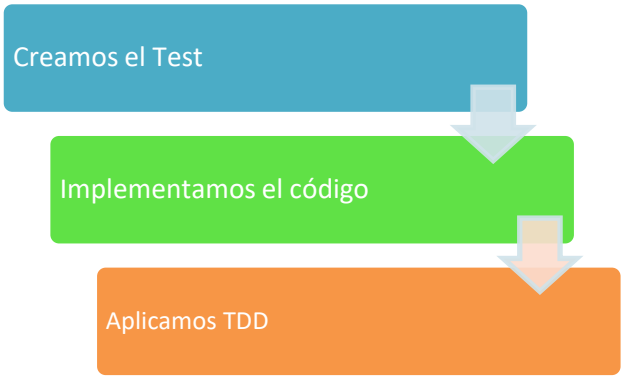

Ilustración 12: Fases de Implementar la funcionalidad con ATDD

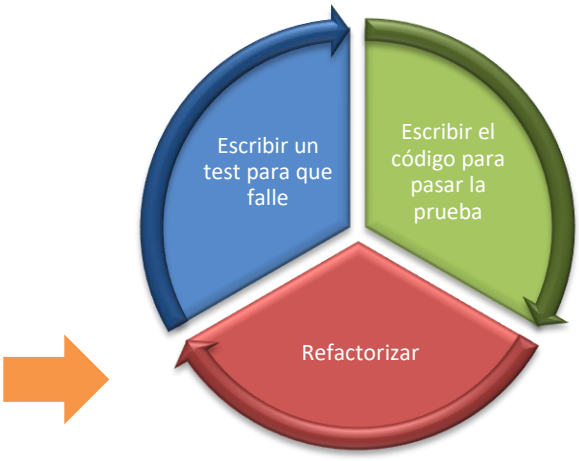

Ilustración 13: TDD

Por último, al final del sprint, se hace una **demostración** al cliente, con la o las funcionalidades creadas.

Las herramientas de software que soportan esta técnica son las mismas que las que usa TDD y algunas de BDD. Aunque ya fueron mencionadas, las más importantes para ATDD son:

- FitNesse (10)
- Robot Framework (23)
- Concordion (97)
- Cucumber (98)

4.2.7. CTDD (Continuous Test-Driven Development)

El Desarrollo Guiado por Test Continuos es una técnica ágil muy reciente y propuesta por dos ingenieros polacos (99). La idea es evitar la pérdida de tiempo que en TDD supone implementar manualmente las pruebas antes del código de la funcionalidad a desarrollar.

Para ello, un programador que aplica TDD para desarrollar software procede a escribir la prueba manualmente para probar la funcionalidad que aún no ha escrito. Y en vez de ejecutar la prueba como haría según el procedimiento de TDD, el test se ejecutará automáticamente por un IDE y se proporciona automáticamente el feedback al programador. Teniendo esto en cuenta, ahora el programador podrá implementar el código de la funcionalidad mientras se ejecutan pruebas en segundo plano automáticamente gracias al IDE.

Esto se puede realizar gracias a un plugin llamado AutoTest.NET4CTDD, creado para Visual Studio. El plugin supervisa la implementación del código automáticamente, eliminando así la ejecución manual del test en la primera fase de TDD.

Existen herramientas de pruebas para habilitar la integración de pruebas continua, como por ejemplo Parasoft Development Testing Platform (100), pero no existe para habilitar la integración continua sobre TDD.

Como es una técnica muy reciente el plugin es limitado para dicho IDE. Actualmente están integrando el plugin en Eclipse, otro IDE muy utilizado para desarrollar software.

5. CASO DE ESTUDIO PRACTICO

En este apartado se detalla la realización de un caso de estudio, aplicando los conceptos desarrollados en este TFG.

5.1. MOTIVACIÓN

Este estudio se ha realizado con el objetivo de comprobar el uso de metodologías de desarrollo de software en diversas empresas de nuestro entorno (Extremadura, España) y comprobar qué metodologías usan y en el caso de utilizar metodologías ágiles, si utilizan pruebas ágiles o no, entre otros datos de interés. Un trabajo teórico, donde se estudien las metodologías existentes, sus mecanismos y las herramientas más utilizadas disponibles es una buena presentación sobre las pruebas ágiles, pero para este trabajo fin de grado hemos querido ir más allá y comprobar qué métodos de trabajo utilizan las empresas.

La finalidad del estudio es hacer un informe sobre cómo trabajan las empresas y comprobar de primera mano el uso de las prácticas ágiles en entornos reales. Para el caso de estudio, si una empresa no utilizase pruebas ágiles, no importa, porque también se comprueba la forma en la que desarrollan sus productos. Lógicamente, no todas las empresas son iguales y hay de tener en cuenta el presupuesto y el número de trabajadores que tiene, así como el número de proyectos simultáneos que es capaz de soportar.

El planteamiento de la necesidad de este tipo de pruebas es importante y es un aspecto que, actualmente tiene una gran relevancia en el sector y por ello, se ha querido hacer un estudio para comprobar el uso de estas técnicas.

Desde mi punto de vista, es interesante hacer el estudio práctico, cuantificando con datos reales el uso de las técnicas descritas en este documento. El espacio muestral del estudio es limitado y está acotado a la región de Extremadura por dos motivos. El primero es la facilidad de contactar y entrevistar al personal de empresas que, en particular, están en un entorno próximo a la Universidad de Extremadura, y el segundo es la no disposición de más tiempo para evaluar a más empresas, sobre todo, aquellas que se ubican más alejadas.

5.2. EL SECTOR TIC EN EXTREMADURA

Para contextualizar este estudio práctico, se presenta a continuación una serie de datos relacionados con el sector TIC en Extremadura.

Vamos a analizar el número de empresas existentes en Extremadura en cuatro sectores: industria, construcción, servicios y TIC. En ese sentido hemos obtenido que existen 4.749 dedicadas a la industria, 8.395 dedicadas a la construcción, 29.330 empresas dedicadas a servicios y 828 empresas TIC (Ilustración 13).

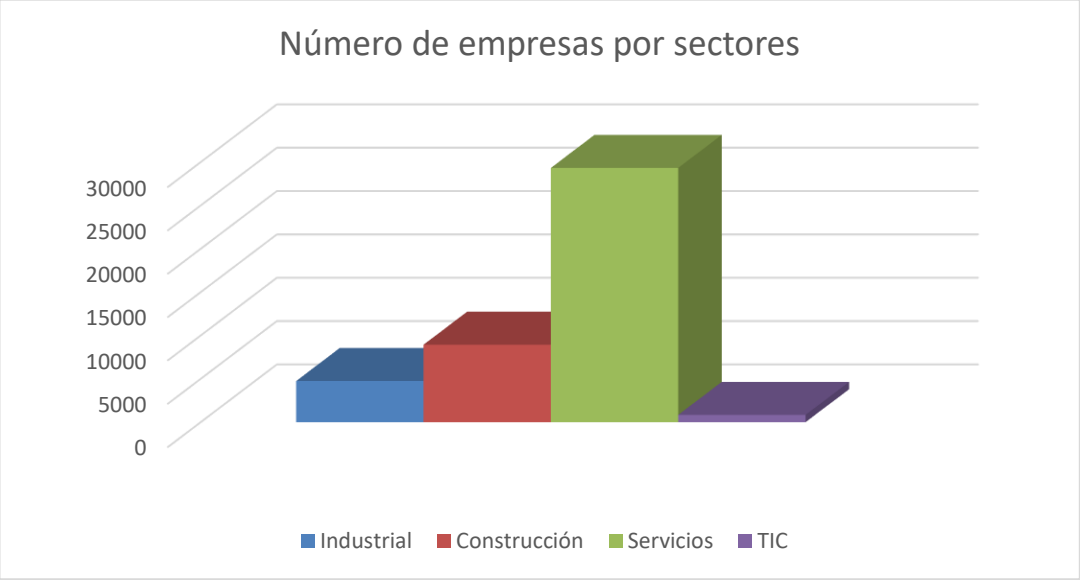

Ilustración 14: Número de empresas por sectores

Actualmente, de esas 828 (101) empresas TIC en Extremadura, 338 pertenecen a la categoría de "Diseño, Desarrollo e Implementación" de productos que pueden ser aplicaciones de escritorio o servidores, portales web o software orientado a internet, aplicaciones multimedia, aplicaciones para dispositivos móviles o videojuegos. A modo de curiosidad de todas estas empresas sólo 69 están ubicadas en la provincia de Cáceres.

Todas las empresas TIC están agrupadas en el sector servicios, sector que supone aproximadamente el 60% del PIB extremeño. No obstante, la aportación al PIB de las empresas TIC sólo llega a un 1,84% (102).

Para finalizar este apartado me gustaría comprobar en qué situación se encuentran las empresas de desarrollo de software en Extremadura con respecto a otras comunidades autónomas.

Para ello, se muestra en la siguiente ilustración que muestra el porcentaje respecto al PIB de cada comunidad que aporta su sector TIC (102).

	2010	2011	2012	2013	2014	2015
Andalucía	2,7	2,7	2,7	2,7	2,5	2,4
Aragón	2,8	2,8	2,8	2,7	2,7	2,6
Asturias	2,8	2,8	3,0	3,1	3,0	2,8
Baleares	2,9	2,8	2,7	2,6	2,4	2,3
Canarias	3,2	3,1	3,0	3,0	2,8	2,7
Cantabria	2,7	2,6	2,6	2,7	2,5	2,5
Castilla y León	2,2	2,2	2,1	2,1	2,0	1,9
Castilla La Mancha	2,2	2,1	2,1	2,1	2,1	2,0
Cataluña	4,1	4,1	4,1	4,1	3,9	3,8
Comunidad Valenciana	2,7	2,6	2,6	2,6	2,4	2,3
Extremadura	2,4	2,3	2,2	2,1	2,0	1,8
Galicia	2,8	2,7	2,8	2,6	2,5	2,4
Comunidad de Madrid	10,4	10,4	10,5	10,7	10,6	10,3
Murcia	2,4	2,3	2,2	2,2	2,0	1,9
Navarra	2,5	2,4	2,3	2,3	2,1	2,0
País Vasco	3,4	3,3	3,3	3,3	3,2	3,1
La Rioja	2,0	2,0	2,0	1,9	1,9	1,8
Ceuta	1,2	1,1	1,1	1,0	1,0	1,0
Melilla	0,8	0,8	0,8	0,8	0,8	0,8

Ilustración 15: Aportación al PIB del sector TIC por comunidades (102)

Exceptuando Ceuta y Melilla se observa cómo el porcentaje de aportación al PIB extremeño es bajo y se contrae cada año y teniendo en cuenta que las empresas casi en su totalidad son pequeñas deben sacar el máximo partido a sus trabajadores utilizando una metodología que saque adelante el producto con la mayor eficacia posible.

5.3. CONTACTO Y OBTENCIÓN DE LOS DATOS

Para realizar el informe, lo primero que se ha realizado ha sido un proceso de recolección de datos. Para ello hicimos entrevistas al personal de las empresas seleccionadas, para poder obtener datos fiables para el estudio.

La primera fase consiste en escoger las empresas que serán las referencias para el estudio. Después contactamos con ellas e hicimos la petición de una entrevista con el responsable de calidad de proyectos; a las empresas que accedieron, les hicimos la entrevista formal.

Para realizar la entrevista diseñamos un modelo único, con el objetivo de seguir el mismo formato en todas ellas. En él se abordan cuestiones relacionadas con el trabajador a entrevistar, su empresa y la metodología que siguen para desarrollar producto. Finalmente, se pregunta por las pruebas. Tras determinar qué preguntas son relevantes se diseñó, además de la entrevista, un formulario de apoyo que serviría para obtener más datos y de forma más concreta.

Cuando contactamos con el trabajador y se realizó la entrevista, se pocedió de la siguiente manera: en primer lugar, se le agradeció su participación. Después se le informó del motivo de la entrevista y se le detalló el contenido del estudio. Por último, se le describió el funcionamiento de la entrevista: grabación de datos, duración etc.

Las preguntas de la entrevista son las siguientes:

1. ***¿A qué se dedica en la empresa?***
2. ***¿Qué tipos de proyectos realizan?***
3. ***¿Los desarrollos son completos o son productos que luego se exportan (SDK) a otros desarrolladores?***
4. ***¿Cuántas personas suelen participar en un proyecto?***
5. ***¿Cómo es la jerarquización de los grupos de los proyectos?***
6. ***¿Qué tipo de ciclo de vida o desarrollo utilizáis?***
7. ***Respecto a las pruebas ¿utilizáis un equipo de testing distinto de los programadores? ¿o bien son los propios programadores los que prueban el código?***
8. ***¿Las pruebas son una fase más del desarrollo o están integradas en otras fases?***
9. ***¿Cuál es vuestra filosofía de pruebas de software?***
10. ***¿Cuándo es la primera vez, en el proyecto, que se habla acerca de las pruebas?***
11. ***¿Hacéis alguna práctica de agile testing o son pruebas tradicionales?***
12. ***¿Hay algún método concreto para hacerlas?***
13. ***¿Habéis cambiando el método de pruebas en estos últimos 5 años?***
14. ***¿Tienen las pruebas ágiles algún incremento en el presupuesto respecto a desarrollo anteriores con pruebas tradicionales?***

Mientras se va preguntando al trabajador, se rellena el formulario de apoyo. El contenido del formulario se ha agregado como anexo y se encuentra al final de este TFG (Anexo IV). En él se cuantifican los datos referidos en las preguntas.

Tras realizar la entrevista con un trabajador de una empresa, se procede a concertar otra entrevista en otra empresa diferente. Los datos obtenidos se almacenan y se van procesando.

Además de las entrevistas personales, y con el fin de ampliar el estudio se optó por utilizar una encuesta. Se creó un formulario web y se envió a varias empresas. De esta manera se obtendría vía web más datos para el estudio. El formulario consta de 17 preguntas que analizan, entre otros aspectos, el tamaño de la empresa, la forma de trabajar y su opinión respecto a las metodologías de desarrollo que utilizan.

Está basado en el formulario de apoyo a las entrevistas, y por lo tanto ambos son muy similares. Esto nos permite mantener el formato único de entrevista para todas las empresas.

De las 68 empresas con las que se contactó 17 accedieron a participar en el estudio.

5.4. REALIZACIÓN DEL ESTUDIO

De las empresas participantes en el estudio se han podido extraer los siguientes datos:

En primer lugar, se extrajo información acerca de cómo son las empresas que participan en este estudio.

5.4.1. Tamaño de las empresas participantes en el estudio

De las empresas participantes del estudio, el 64% afirmó tener 5 trabajadores o menos, el 28% tiene en nómina entre 6 y 19 trabajadores y el resto (sobre un 7%) más de 20 trabajadores. Este dato indica que 6 de cada 10 empresas TIC en Extremadura son pequeñas o muy pequeñas en función de sus recursos humanos (Ilustración 14).

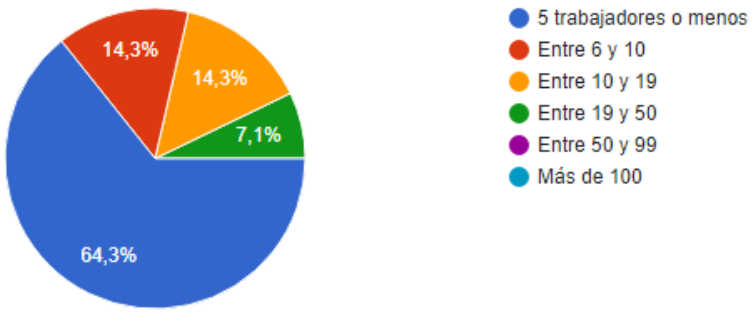

Ilustración 16: Número de trabajadores de las empresas

5.4.2. Capacidad para abordar proyectos o líneas de negocio simultáneas

En la línea de la respuesta anterior, el 43% de las empresas indicó que tenían una capacidad de soportar entre 2 y 3 proyectos a la vez, el 36% entre 3 y 6 proyectos y sólo el 14% más de 7 proyectos a la vez. Esto nos da una idea de lo habitual que resulta encontrar en las empresas extremeñas entre 3 y 6 líneas de negocio a la vez. Esto indica que de media estarán en producción unos 5 productos por empresa TIC en Extremadura (Ilustración 15).

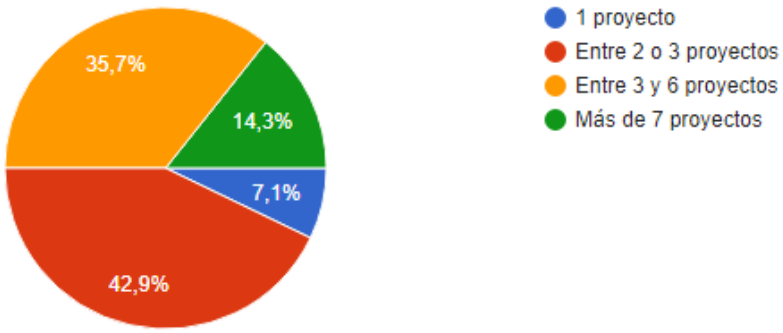

Ilustración 17: Número de líneas de negocio que soportan las de las empresas

5.4.3. Personal asignado por proyectos

Respecto al personal asignado por proyectos, la respuesta es contundente: el 93% afirma que un equipo de trabajo está formado por 5 trabajadores o menos, estando dedicados al desarrollo, total o parcial, de un producto. Este dato deja entrever qué tipo de metodologías emplean y qué métodos usarán para organizarse. Claramente, al tener equipos tan pequeños, se deben usar metodologías ágiles para su organización. Es el tipo de metodología idóneo para trabajar en equipos de estas características.

El siguiente bloque de información a obtener se refiere a los métodos de trabajo que utilizan las empresas participantes en el estudio y tasa de éxito por proyecto.

5.4.4. Metodologías utilizadas

El 79% de las empresas analizadas utilizan la metodología Ágil para desarrollar software mientras que el resto utilizan una metodología en cascada. Sin embargo, nos ha llamado la atención otro dato que muestra que todas las empresas encuestadas han usado alguna vez algún tipo de metodología ágil.

Dentro de los tipos de metodologías ágiles el 90% afirmó usar o haber usado Scrum, un 50% Kanban, un 29% XP (Extreme Programming), y sólo un 14% LD (Lean Development).

5.4.5. Tasa de éxito por proyectos

Respecto a la tasa de éxitos por proyectos propusimos que nos indicaran con un 100% si todos sus proyectos acababan en una satisfacción absoluta del cliente; un 75% si algún proyecto excepcional fue mal; con un 50% si la mitad de sus proyectos acaban en éxito; o un 25% si la mayoría de sus proyectos no terminaban en producción.

Las respuestas nos indican que el 28% de las empresas acababan con todos sus desarrollos exitosos, un 64% indico que tuvieron éxito en la mayoría de sus desarrollos, excepto algún caso excepcional de fracaso, y sólo un 7% dijo tener la mitad de sus proyectos con éxito (Ilustración 16).

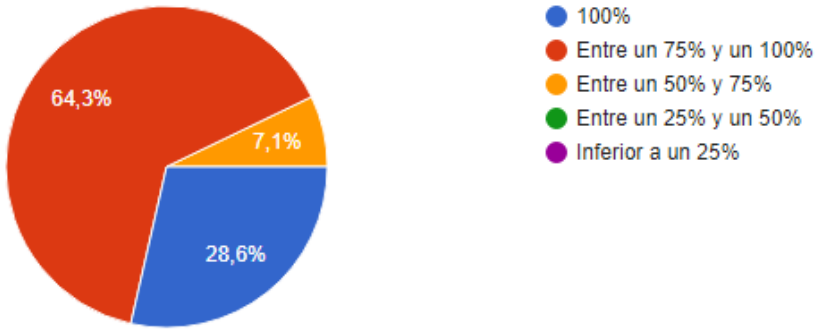

Ilustración 18: Tasa de éxitos de las empresas

5.4.6. Tipos de prueba de las empresas participantes en el estudio

Al preguntar por el tipo de pruebas que realiza cada empresa distinguimos entre: pruebas ágiles, donde las pruebas están integradas en todas las fases del desarrollo, o pruebas tradicionales, donde las pruebas son una fase más del desarrollo.

El resultado fue que el 64% de las empresas realizan pruebas ágiles frente a un 35% que realiza pruebas tradicionales. De las empresas que realizaban pruebas ágiles, la mayoría, el 50%, realizaba pruebas específicas de la empresa, el 28% realizaban TDD y el 20% BDD.

Para realizar las pruebas, el 60% de las empresas opta por una estrategia donde los programadores tienen ayuda de otros miembros del equipo para realizar tareas de

testing; un 30% son los programadores los únicos encargados de realizar las pruebas y el resto usa un método ágil para hacer testing (testing colaborativo, integración continua TDD, BDD o ATDD).

5.4.7.Fase en la que se habla del testing por primera vez en un desarrollo

Otra pregunta interesante para nosotros era conocer en qué fase se empezaba a hablar de las pruebas. El 73% de las empresas empieza a pensar en las pruebas en las fases de implementación; un 26% en la fase de diseño y sólo un 13% en fases tempranas del desarrollo como la definición y análisis de requisitos.

5.4.8.Tiempo dedicado a las pruebas

Otro aspecto que consideramos era relativo al tiempo que dedican las empresas a las pruebas, y el impacto económico de éstas en el desarrollo del proyecto. Como es sabido, las pruebas deberían ocupar una parte importante del tiempo empleado en desarrollar un producto, pues esto provocará un aumento de la calidad del producto final.

El resultado del estudio nos dice que 14% de las empresas estudiadas dedican entre un 40% y un 60% del tiempo empleado para desarrollar un producto a pruebas. No obstante, el 80% de las empresas restantes se dividen entre: un 40% que dedica entre un 20% y un 40% del tiempo del desarrollo a pruebas; y otro 40% de las empresas que sólo dedica entre un 10% y un 20% de su tiempo de desarrollo a pruebas (Ilustración 17).

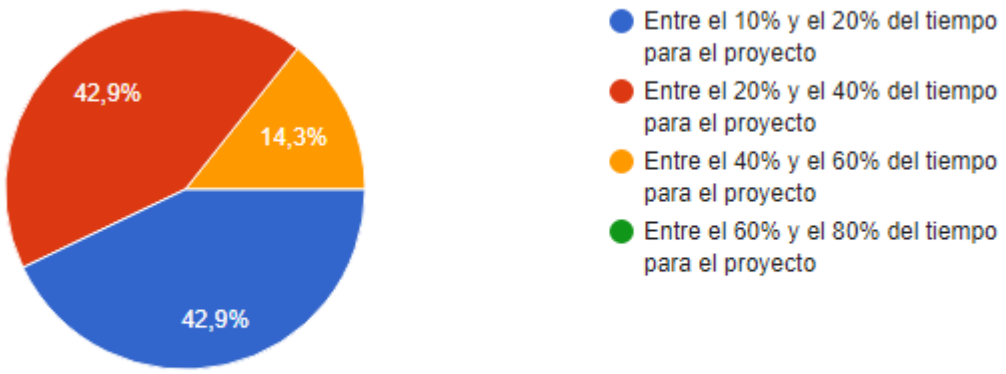

Ilustración 19: Tiempo invertido en las pruebas de las empresas

5.4.9. Impacto económico de las pruebas en el proyecto

Por otra parte, y como se puede apreciar en la ilustración 18, para un 61% de las empresas, las pruebas suponen un gasto que representan menos de un 25% de los recursos económicos dedicados al proyecto software. El 23% de ellas afirman que las pruebas les suponen un gasto de entre un 25% y un 50% del presupuesto asignado al desarrollo y el 7% restante que afirma que les supone más de un 50% del presupuesto asignado al desarrollo.

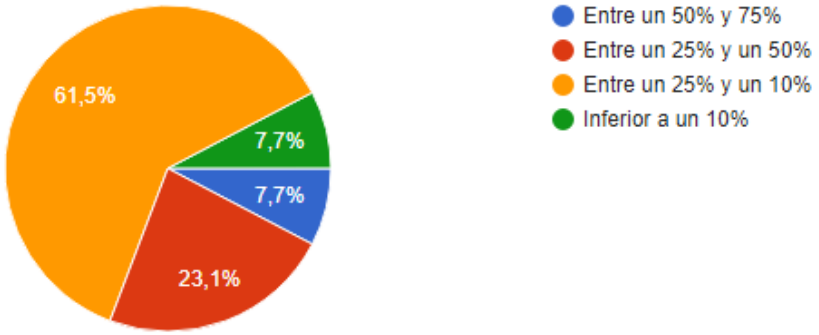

Ilustración 20: Gasto expresado en % del presupuesto del proyecto en las pruebas de las empresas

5.4.10. Diferencia entre aplicar pruebas ágiles y pruebas tradicionales en eficacia

En el último bloque de la recogida de datos consistió en una comparación entre pruebas ágiles y tradicionales.

Al preguntar sobre la opinión, basada en la experiencia que han tenido en pruebas ágiles respecto a la eficacia en la detección de errores, frente a las pruebas tradicionales, para un 46%, las pruebas ágiles son eficaces en la mayoría de los casos; un 23% afirma con rotundidad que usar prácticas ágiles es siempre más eficaz frente a un 31% que afirma no haber notado una diferencia significativa respecto a la eficacia entre pruebas ágiles o pruebas tradicionales (Ilustración 19).

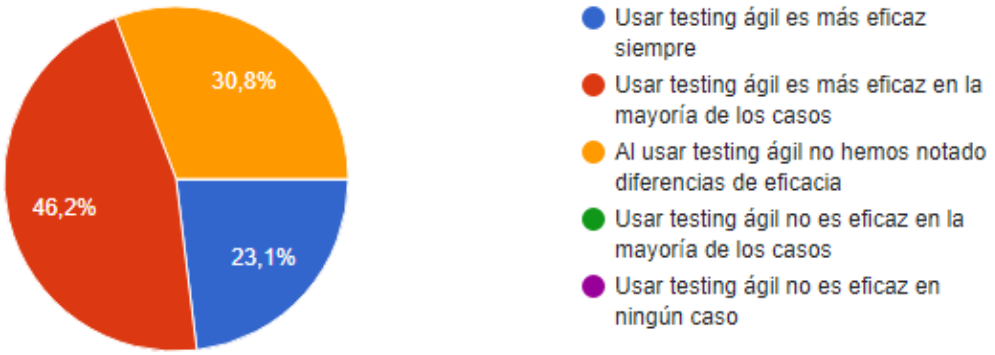

Ilustración 21: Eficacia del agile testing para las empresas

5.4.11. Diferencia entre aplicar pruebas ágiles y pruebas tradicionales en tiempo

Otra cuestión interesante relacionada con la anterior es la diferencia entre pruebas ágiles y tradicionales en tiempo invertido en ellas. Las respuestas son diversas. Por un lado, obtuvimos que el 38% de las empresas afirmaron que usar pruebas ágiles ahorra tiempo en la mayoría de los casos y un 31% afirmó que ahorraban tiempo, en cualquier caso. Por otro lado, sólo un 21% de las empresas dijo no haber notado diferencia de tiempo o que no ahorraba tiempo en la mayoría de los casos. Por último, un 7% afirmó que el agile testing producía una pérdida de tiempo respecto a las pruebas tradicionales (Ilustración 20).

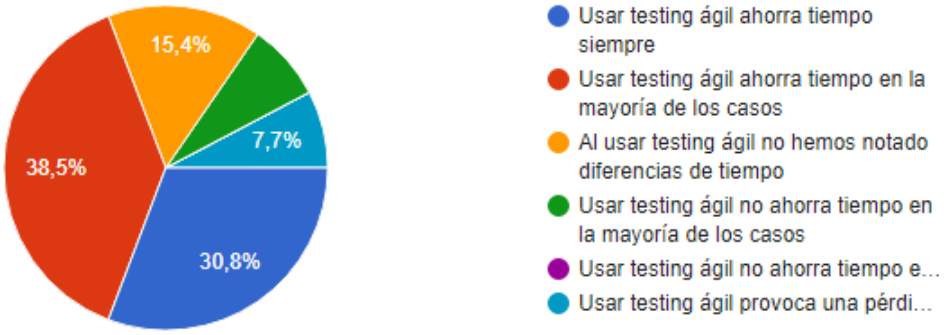

Ilustración 22: Impacto en el tiempo de desarrollo según las empresas

5.4.12. Diferencia entre aplicar pruebas ágiles y pruebas tradicionales en el presupuesto

El último dato extraído del estudio se refiere a la diferencia entre pruebas ágiles y tradicionales en el impacto presupuestario del desarrollo. Sólo el 38% de las empresas afirmó que las pruebas ágiles ahorraban dinero o eran más baratas en la mayoría de los casos, frente a otro 38% que afirma que, en relación a las pruebas ágiles o bien no habían notado una diferencia o eran más caras en la mayoría de los casos. El dato más importante es que junto a ese 38% hay que considerar un 23% de las empresas que afirma que "las pruebas ágiles no ahorran dinero en ningún caso y de hecho incrementa el presupuesto del desarrollo de software".

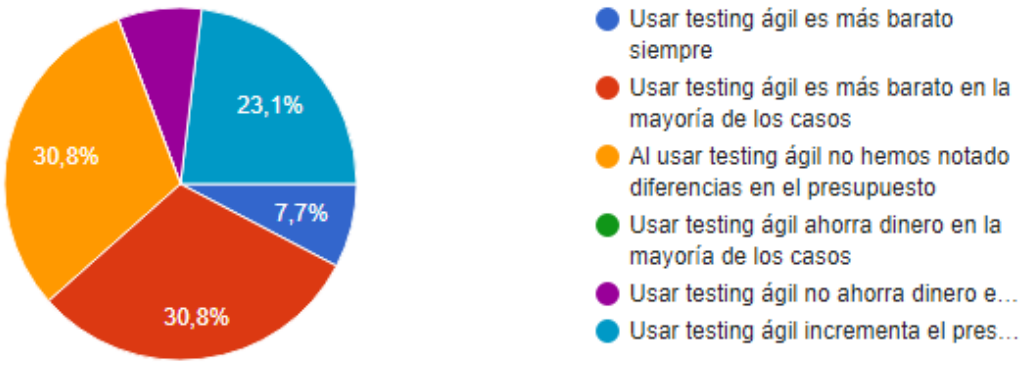

Ilustración 23: Impacto en el presupuesto según las empresas

6. DISCUSIÓN Y CONCLUSIONES

En este capítulo se explica las conclusiones obtenidas en base a los datos recogidos en el estudio explicado en el capítulo anterior. En este estudio se ha podido determinar las siguientes conclusiones:

- El sector empresarial TIC en Extremadura está compuesto casi en su totalidad por empresas pequeñas. Podemos afirmar que aproximadamente 8 de cada 10 empresas extremeñas tiene menos de 10 trabajadores.
- Estas empresas mantienen una media de unos 4 productos o líneas de negocio. Por lo tanto, cada producto software desarrollado en Extremadura está mantenido por unos 3 trabajadores.
- Respecto a las metodologías utilizadas en la actualidad 8 de cada 10 empresas en Extremadura utilizan metodologías ágiles, y el resto se decantan por estrategias iterativo e incremental o cascada. La metodología ágil más usada es Scrum en 9 de cada 10 empresas, siguiendo a esta Kanban, pero a una distancia considerable.
- La tasa de éxitos por proyecto es muy alta superando, el 75% de éxitos por proyectos realizados
- Una mayoría de las empresas utilizan pruebas ágiles para testear sus productos, en concreto, 6 de cada 10 empresas, siendo TDD y BDD las prácticas más utilizadas.
- Del estudio se desprende que el uso de pruebas ágiles no ahorra tiempo respecto a las prácticas tradicionales, y que normalmente se suele incrementar el tiempo de desarrollo de un producto al usar alguna técnica ágil. Esto trae consigo un incremento del coste al desarrollar un producto. No obstante, este gasto es absorbido en las fases de mantenimiento, ya que elimina muchas acciones a realizar en esta fase. Por lo tanto, se incrementa el desarrollo, pero se abarata el mantenimiento.
- La eficacia en las pruebas ágiles es superior a las pruebas tradicionales.
- Los productos que usan *agile testing* salen con mayor calidad respecto a las pruebas tradicionales.

Con estos datos podemos concluir que hay un porcentaje muy amplio de empresas en Extremadura que usan pruebas ágiles y coincide que todas las que usan este tipo de pruebas son empresas pequeñas, con menos de 10 trabajadores. En este tipo de empresas se observa que, aplicando pruebas ágiles, se consigue una mayor calidad en el resultado final de un proyecto software. Así lo afirman las empresas encuestadas que usan este tipo de pruebas, aunque reconocen el aumento del tiempo en los desarrollos.

Estas prácticas aplicadas a empresas pequeñas, consiguen que el proyecto esté más enfocado a minimizar el impacto de los fallos desde las fases tempranas del desarrollo. Las empresas más grandes, superiores a 10 trabajadores, se muestran más perezosas al cambio. No obstante, tienen un músculo económico superior, y eso les permite tener mejores recursos para detectar fallos con pruebas tradicionales.

Otro aspecto interesante es investigar por qué se aumenta la calidad de la prueba. Para las empresas pequeñas es muy ventajoso hacer énfasis en desarrollar código con el menor número de fallos posibles, desde el comienzo del proyecto. La clave reside en la diferencia con las pruebas tradicionales.

En el agile testing se tiene un desarrollo que ha sido guiado por las pruebas, enfocado a no cometer errores, los posibles errores están localizados y son fácilmente aislables permitiendo una localización más rápida. Mientras que, en las pruebas tradicionales, hay que realizar baterías exhaustivas de pruebas (unitarias, de sistemas etc.) para ir en busca de posibles errores en cualquier parte del código. Esto no quiere decir que no haya que realizar pruebas unitarias, por ejemplo, en el testing ágil. Hay que hacerlas, por su puesto, pero a la hora de localizar un error, hay que entender que el código se desarrolló en base a no cometer fallos básicos y es más fácil encontrar errores gracias, a la estructura y forma que tiene el código generado después de pensar que va a fallar y cómo se va a programar el código para que no falle.

Respecto a la tasa de éxitos por empresa, se observa un porcentaje elevado de éxitos con respecto la tasa que recoge el informe CHAOS (5) comentado en capítulos anteriores. Entendemos que ese porcentaje de éxitos es tan elevado es debido a que la mayoría de las empresas cuentan con un presupuesto limitado y deben volcar todos sus esfuerzos para sacar los proyectos adelante. Este dato sumado al que indica que el

60% de las empresas que usan metodologías ágiles usan prácticas ágiles, indica que el agile testing es para la mayoría una buena manera de invertir recursos para aumentar la calidad del producto final y obtener éxito en la fase de producción.

Como conclusión al estudio, podemos decir que en el caso de las empresas extremeñas y según indica el estudio, lo habitual es el uso de Scrum y es raro el uso de otras metodologías. Esto se debe a la composición de la empresa extremeña. Scrum favorece la productividad y es eficaz en equipos pequeños. Si tenemos en cuenta que cada empresa dedica unos 3 trabajadores a cada producto, observamos que esta adopción tiene mucho sentido.

La necesidad de ser productivos y alcanzar el éxito ha hecho que las empresas busquen formas de trabajar eficientes. Las pruebas ágiles suponen para más de la mitad de las empresas extremeñas una forma de alcanzar la calidad en el producto final. Aunque se ha comprobado que incrementa el presupuesto y el tiempo del proyecto, a la larga, evita quebraderos de cabeza y ahorro de dinero y tiempo en las fases de explotación y mantenimiento.

Como conclusión final, me gustaría señalar en primer lugar el buen estado de eficiencia de las empresas en Extremadura que con unos recursos limitados son capaces de generar productos con un alto grado de calidad, permitiendo ser competitivos con respecto a otras empresas situadas en otras comunidades. El estudio ha aportado datos muy interesantes no sólo sobre la composición del sector TIC dedicado al desarrollo de software, sino que también se han aportado datos acerca de cómo trabajan y lo más importante, para nuestro objetivo, muestra el uso de las pruebas ágiles en las empresas de Extremadura.

Se determina, por tanto, que el sector de TIC en Extremadura está compuesto por empresas pequeñas, que en su mayoría aplican metodologías ágiles siendo SCRUM la metodología usada por defecto. Las empresas más grandes (por número de trabajadores) suelen aplicar también SCRUM, aunque por otra parte, también se utiliza muchas metodologías propias de las empresas en iterativo e incremental. También se llega a la conclusión de que hay una mayoría de empresas que usan prácticas ágiles para ejecutar las pruebas, y que todas ellas, afirman que utilizándolas han notado un aumento en la calidad de sus productos. También han mostrado un lado amargo, y es la dilatación en el tiempo de los desarrollos de sus proyectos al usar prácticas ágiles,

llevando al aumento del presupuesto de los desarrollos. No obstante, este incremento económico se ve amortizado en las fases de mantenimiento y explotación, ya que al salir el producto a producción con mayor calidad hay menos fallos que reparar. El cliente lógicamente, experimenta un mayor grado de satisfacción que lleva a elevar sus porcentajes de éxitos absolutos.

REFERENCIAS BIBLIOGRÁFICAS

1. **RedBull Mexico.** RedBull Mexico. *RedBull Mexico.* [En línea] RedBull News, 27 de Abril de 2015. [Citado el: 20 de 9 de 2016.] http://www.redbull.com/mx/es/games/stories/1331719692857/%C2%BFsabes-por-qu%C3%A9-se-cancel%C3%B3-silent-hills.

2. **Khurana, Rohit.** *Software Engineering.* Jangpura, New Delhi : Vikas® Publishing House Pvt. Ltd., 2006.

3. **Standish Group.** *The CHAOS report.* Boston, Massachusetts, EEUU : Standish Group Internacional, Inc., 1994.

4. **ISO.** *Norma ISO 9000:2000.* EEUU, 15 de 12 de 2000. Software.

5. **Shane Hastie, Stéphane Wojewoda.** Standish Group 2015 Chaos Report - Q&A with Jennifer Lynch. *Standish Group 2015 Chaos Report - Q&A with Jennifer Lynch.* [En línea] Standish Group , 4 de Ocutubre de 2015. [Citado el: 13 de 11 de 2016.] https://www.infoq.com/articles/standish-chaos-2015.

6. **de Antonio, Angélica.** valoryempresa.com. *valoryempresa.com.* [En línea] 25 de 3 de 2005. [Citado el: 6 de 10 de 2016.] http://valoryempresa.com/archives/calsoftware.pdf.

7. **Pérez Toledano, M.A y Navasa , A.** Tema 6: Técnicas de Control de Calidad de un Sistema. [aut. libro] Universidad de Extremadura. *Ingeniería del Software.* Cáceres : s.n., 2016.

8. **Polo Usaola, Macario, Pérez Lamancha, Beatriz y Reales Mateo, Pedro.** *Técnicas ombinatorias y de mutación para testing de sistemas software.* Madrid : RA-MA, 2012. 978-84-9964-146-1.

9. **Martin Series, Robert C.** *Fit, for developing software.* s.l. : Prentice Hall, 2005. 0321269349.

10. **Adzic, Gojko.** *Test Driven with FitNesse.* London : Neuri Limited, 2009. 978-0-9556836-2-6.

11. **Parasoft.** *Parasoft Service Virtualization and Skytap Cloud.* Monrovia CA, USA : Parasoft Corporation.

12. **Shell, Michael.** *The Testflow User's Guide.* 2009.

13. **XQual.** http://www.xqual.com/products/xstudio.html. *http://www.xqual.com/products/xstudio.html.* [En línea] [Citado el: 01 de 07 de 2017.] http://www.xqual.com/products/xstudio.html.

14. **TRICENTIS Technology & Consulting GmbH.** *Tosca Test Suite.* 2012.

15. **Telerik Software Academy.** *Unit Testing with Mocha and Karma.*

16. **ThroughtWorks.** *Sahi - Web Automation Tool.*

17. **SmartBear.** *TestComplete 12.* 2017.

18. **Soni, Lokesh.** *Automated Test using appium.* [Presentación] s.l. : DRUPAL CON ASIA2016, 2016.

19. **Gargh, Navneesh.** *Test Automation using UFT.* s.l. : AdactIn Group Pty Ltd., 2013. 978-0-9922935-1-2 .

20. **IBM.** *IBM Rational Testing Guide.*

21. **Ranorex.** *Ranorex Using Guide.*

22. **Perforce.** *QA Wizart Pro User Guide.* Minneapolis, Minesota, USA : Perforce Software Inc, 2017.

23. **Robot Framework User Guide.** Robot Framework User Guide. *Robot Framework User Guide.* [En línea] Nokia Network, 2016. [Citado el: 07 de 01 de 2017.] http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html.

24. **Original Software.** Original Software. *Original Software.* [En línea] 2017. https://www.origsoft.com/.

25. **Test Automation Framework.** Test Automation Framework. *Test Automation Framework.* [En línea] Test Automation, 2017. http://menonvarun.github.io/taf/.

26. **Smart Software Testing Solutions Inc.** Opkey. *Opkey.* [En línea] Smart Software Testing Solutions Inc, 2017. https://www.opkey.com/.

27. **Halili, Emily H.** *Apache JMeter.* Olton, Birmingham, UK : Packt Publishing, 2008. 978-1-847192-95-0.

28. **Gatling.** http://gatling.io/. *http://gatling.io/.* [En línea] Gatling, 2017. http://gatling.io/.

29. **Locust.** http://locust.io/. *http://locust.io/.* [En línea] Locust, 2017. http://locust.io/.

30. **IBM.** http://www-03.ibm.com/software/products/es/performance. *http://www-03.ibm.com/software/products/es/performance.* [En línea] 2017. http://www-03.ibm.com/software/products/es/performance.

31. **WAPT.** WAPT. *WAP.* [En línea] 2017. https://www.loadtestingtool.com/.

32. **Automation Anywhere.** https://www.automationanywhere.com/testing. *https://www.automationanywhere.com/testing.* [En línea] https://www.automationanywhere.com/testing.

33. **AppVance.** [En línea] https://www.appvance.com/.

34. **Manage Engine.** [En línea] https://www.manageengine.com/products/qengine/qengine-eol.html.

35. **OpenSTA.** [En línea] http://opensta.org/.

36. **NeoTys.** [En línea] 2017. http://www.neotys.com/.

37. **SOASTA.** [En línea] https://www.soasta.com/load-testing/.

38. **PARASOFT.** PARASOFT LOAD TEST. *PARASOFT LOAD TEST.* [En línea] 2017. https://www.parasoft.com/capability/cloud-testing/.

39. **Hewlett Packard.** Load Runner. *Load Runner.* [En línea] https://saas.hpe.com/es-es/software/loadrunner.

40. **New Relic.** BlazeMeter Load Testing. *BlazeMeter Load Testing.* [En línea] 2017. https://newrelic.com/plugins/blazemeter/2.

41. **DynaTrace.** DynaTrace. *DynaTrace.* [En línea] https://www.dynatrace.com/topics/load-test/.

42. **BrowserStack.** BrowserStack. *BrowserStack.* [En línea] https://www.browserstack.com/.

43. **TestingWhiz.** TestingWhiz. *TestingWhiz.* [En línea] http://www.testing-whiz.com/.

44. **SMARTBEAR.** CrossBrowserTesting. *CrossBrowserTesting.* [En línea] https://crossbrowsertesting.com/.

45. **Inflectra.** Inflectra. *Inflectra.* [En línea] https://www.inflectra.com/Rapise/.

46. **HP.** HP WINRUNNER. *HP WINRUNNER.* [En línea] http://students.depaul.edu/~slouie/wr_tut.pdf.

47. **Canoo.** Canoo We Test. *Canoo Web Test.* [En línea] http://webtest.canoo.com/webtest/manual/WebTestHome.html.

48. **Software Test Tool.** Software Test Tool - VTEST. *Software Test Tool - VTEST.* [En línea] http://sqa.fyicenter.com/FAQ/Testing-Tools/Software_Test_Tools_VTEST.html.

49. **GLOBE.** HP Unified Functional Testing (UFT). *HP Unified Functional Testing (UFT).* [En línea] https://www.globetesting.com/hp-unified-functional-testing-uft/.

50. **Apache Maven.** Easy Mock. *Easy Mock.* [En línea] http://easymock.org/.

51. **Cargo.** Cargo. *Cargo.* [En línea] http://doc.crates.io/guide.html.

52. **SeleniumHQ.** SeleniumHQ. *SeleniumHQ.* [En línea] http://www.seleniumhq.org/.

53. **WET.** WET. *WET.* [En línea] http://www.wet.qantom.org/home.html.

54. **UTest.** UTest. *UTest.* [En línea] https://www.utest.com/tools/watir.

55. **Molyneaux, Ian.** The Art of Performance Testing . *The Art of Performance Testing : Help for Programmers and Quality Assurance.* Sebastopol, CA, USA : O'REILLY, 2009. Vol. 1º Edition. 978-0596520663.

56. **NUnit.** NUnit. *NUnit.* [En línea] https://www.nunit.org/.

57. **JUnit.** JUnit. *JUnit.* [En línea] http://junit.org/junit4/.

58. **TestNG.** TestNG. *TestNG.* [En línea] http://testng.org/doc/.

59. **DBUNIT.** DBUNIT. *DBUNIT.* [En línea] http://dbunit.sourceforge.net/.

60. **XUNIT.** XUNIT. *XUNIT.* [En línea] https://xunit.github.io/.

61. **jMock.** jMock. *jMock.* [En línea] http://www.jmock.org/.

62. **Cgreen.** Cgreen. *Cgreen.* [En línea] https://github.com/cgreen-devs/cgreen.

63. **Parasoft.** Parasoft C/C++ TEST. *Parasoft C/C++ TEST.* [En línea] https://www.parasoft.com/product/cpptest/.

64. **TypeMock.** TypeMock. *TypeMock.* [En línea] https://www.typemock.com/.

65. **AgitarOne.** AgitarOne. *AgitarOne.* [En línea] http://www.agitar.com/solutions/products/agitarone.html.

66. **HttpUnit.** HttpUnit. *HttpUnit.* [En línea] http://httpunit.sourceforge.net/.

67. **SMARTBEAR.** SMARTBEAR Soap UI. *SMARTBEAR Soap UI.* [En línea] https://www.soapui.org/.

68. **POSTMAN.** POSTMAN. *POSTMAN.* [En línea] https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop.

69. **Advance Rest Client.** Advance Rest Client. *Advance Rest Client.* [En línea] https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo.

70. **Appium.** Appium. *Appium.* [En línea] http://appium.io/.

71. **Google.** Google Developer Tools. *Google Developer Tools.* [En línea] https://developer.chrome.com/devtools.

72. **Ruiz-Falcó Rojas, Arturo.** Despliegue de la Función de Calidad. [aut. libro] Arturo Ruiz-Falcó Rojas. *Despliegue de la Función de Calidad, Módulo 8.* Madrid : Universidad Pontificia Comillas ICAI-ICADE, 2009.

73. **GlobalLogic.** *Calidad del Producto en Agile.* [Presentación Power Point] s.l. : GlobalLogic, 2014.

74. **Manifiesto Ágil.** agilemaniefiesto.org. *agilemanifiesto.org.* [En línea] agilemanifiesto.org, 2001. [Citado el: 6 de 10 de 2016.] http://agilemanifesto.org/iso/es/manifesto.html.

75. **Informe Técnico Informático - CES.** Tema 34: Introducción a las Metodologías Ágiles. [aut. libro] Técnico Informático - CES. *Tema 34: Introducción a las Metodologías Ágiles.* 2014.

76. **García Rodrigez, Manuel José.** Estudio comparativo entre metodologías ágiles y las metodologías tradicionales para la gestión de proyectos software. *Estudio*

comparativo entre metodologías ágiles y las metodologías tradicionales para la gestión de proyectos software. Oviedo, Asturias, España. : Univerisdad de Oviedo, 2015.

77. **Hirotaka Takeuchi, Nonaka y Ikujiro.** *The New New Product Development Game.* s.l. : Harvard Business Review, 1986.

78. **Kniberg, Henrik.** *SCRUM Y XP Desde Las Trincheras.* s.l. : C4Media, 2007. 978-1-4303-2264-1.

79. **James, Michael.** http://scrummethodology.com/scrum-effort-estimation-and-story-points/. *http://scrummethodology.com/scrum-effort-estimation-and-story-points/.* [En línea] Learn Scrum, 6 de 10 de 2009. [Citado el: 5 de 11 de 2016.] http://scrummethodology.com/scrum-effort-estimation-and-story-points/.

80. **Hispamedia.** http://hispamedia.es/la-metodologia-scrum/. *http://hispamedia.es/la-metodologia-scrum/.* [En línea] Hispamedia, 7 de 8 de 2014. [Citado el: 05 de 11 de 2016.] http://hispamedia.es/la-metodologia-scrum/.

81. **Beck, Kent.** *Extreme Programming Explained.* Stoughton, Massachusetts : Pearson Education, Inc., 2005. 0-321-27865-8.

82. *The New XP.* **Marchesi, Michele.** 2005.

83. **Nöteberg, Staffan.** *Pomodoro Technique Illustrated.* s.l. : Pragmatic Programmers, LLC., 2009. 978-1-934356-50-0.

84. **Joskowicz, José.** *Reglas y Prácticas en Extreme Programming.* Vigo, España : Universidad de Vigo, 2008.

85. **Crispin, Lisa y Gregory, Janet.** Agile Testing: A Practical Guide for Testers and Agile Teams . *Agile Testing: A Practical Guide for Testers and Agile Teams* . Boston, USA. : Addison-Wesley, 2009. 9788131730683.

86. **SpectFlow.** SpectFlow. *SpectFlow.* [En línea] http://specflow.org/.

87. **Gradle .** Gradle TestKit. *Gradle TestKit.* [En línea] https://docs.gradle.org/3.3/userguide/test_kit.html.

88. **TestPlant.** eggPlant. *eggPlant.* [En línea] https://www.testplant.com/eggplant/testing-tools/.

89. **ThoughtWorks.** Twist. *Twist.* [En línea] https://www.thoughtworks.com/products/twist-agile-testing.

90. *Using Behavioral Driven Development (BDD) in Capstone Design Projects.* **Goulart, Ana Elisa E.** 10702, Texas : 360 of Engineering Education, 2014.

91. **Dobinson, Tee.** *The Gherkin Guide.* s.l. : BAIZDON, 2013. 978-0957401204.

92. **University of North Florida.** *Chapter 4 – Requirements Engineering.* 2013.

93. **Calabash.** Calabash. *Calabash.* [En línea] http://calaba.sh/.

94. **Codeception.** Codeception. *Codeception.* [En línea] http://codeception.com/.

95. **Froglogic.** Froglogic. *Froglogic.* [En línea] https://www.froglogic.com/squish/.

96. **TestingWithFrank.** TestingWithFrank. *TestingWithFrank.* [En línea] http://www.testingwithfrank.com/.

97. **Concordion.** Concordion. *Concordion.* [En línea] http://concordion.org/.

98. **Wynne, Matt.** *The Cucumber Book.* s.l. : Pragmatic Programmers, LLC, 2012.

99. *Continuous Test-Driven Development, A Novel Agile Software.* **Madeyski, Lech y Marcin Kawalerowicz.** Breslavia : Wroclaw University of Technology, 2013.

100. *Modern Application Functional Test Automation Tools.* **Giudice, Diego Lo.** Cambridge, MA, USA. : Forrester, 2016.

101. **Computaex.** Observatorio regional de información del Sector TIC. *Observatorio regional de información del Sector TIC.* [En línea] Junta de Extremadura, 2017. [Citado el: 20 de 06 de 2017.] https://olistic.cenits.es/listado-de-empresas?field_entidad_actividades_tic_tid%5B%5D=24&title=&field_entidad_localidad_value=&field_entidad_provincia_value=All&field_entidad_constitucion_value=&field_entidad_disolucion_value=&field_entidad_pura_tic_value.

102. **Fundación COMPUTAEX.** *TaxonomTIC-2016: Situacion de un sector clave en Extremadura.* Cáceres, Extremadura, España : s.n., 2016.

103. **Martin C. Robert, Martin Micah.** *Agile Principles, Patterns, and Practices in C#.* s.l. : Prentice Hall, 2006. 978-0-13-185725-4.

104. **Jurado, Carlos Blé.** *Diseño Ágil con TDD.* s.l. : safeCreative, 2010. 978-1-4452-647-4.

105. **Chelimsky, David.** *The RSpec Book: Behaviour-Driven Development.* Dallas, Texas, USA : The Pragmatic Bookshelf, 2012. 978-1-934356-37-1.

106. **Gartner, Markus.** *ATDD by Example: A Practical Guide to Acceptance.* Westford, Massachusetts, USA : Pearson Education, Inc., 2013. 978-0-321-78415-5.

107. **Ahsan Nawaz, Kashif Masood Malik.** *SOFTWARE TESTING PROCESS IN AGILE DEVELOPMENT.* Ronneby, Sweden : Department of Interaction and System Design Blekinge Institute of Technology.

108. **Wikipedia, La Enciclopedia Libre.** Wikipedia, La Enciclopedia Libre. *Wikipedia, La Enciclopedia Libre.* [En línea] Wikipedia, La Enciclopedia Libre, 24 de 6 de 2010. [Citado el: 10 de 10 de 2016.] https://es.wikipedia.org/wiki/F1_2010_(videojuego).

109. **Dancirocco.** web.archive.org. *web.archive.org.* [En línea] Internet Archive, Wayback Machine, 23 de 9 de 2010. [Citado el: 10 de 10 de 2016.] http://web.archive.org/web/20101027145135/http://community.codemasters.com/forum/f1-2010-game-1316/429298-unofficial-f1-2010-bugs-errors-list.html.

110. **Fernández, Salva.** Meristation. *Meristation.* [En línea] Grupo PRISA, 9 de 10 de 2016. [Citado el: 10 de 10 de 2016.] http://www.meristation.com/noticias/mafia-iii-recibe-parche-para-desbloquear-el-framerate/2152027.

111. **ISO.** ISO. *http://www.iso.org/iso/home.htm.* [En línea] 1 de 2 de 2008. [Citado el: 10 de 10 de 2016.] http://www.iso.org/iso/catalogue_detail?csnumber=43447.

112. **colaboradores, Carlos Ble Jurado y.** *Diseño Ágil con TDD.* Madrid : SafeCreative, 2010. 9781445264714.

113. **Martínez, David Luis la Red.** *Sistemas Operativos.* Corrientes (Argentina) : UNNE, 2001.

114. **Caballeira, Félix García.** *Sistemas Operativos: Una Visión Aplicada.* Madrid : MCGRAW-HILL INTERAMERICANA DE ESPAÑA, 2001. 84-481-3001-4.

115. **Esteban, Paulo Clavijo.** *ATDD Desarrollo Dirigido por Test de Acepatación.* [Diapositivas] 2013.

116. **Fontela, Carlos.** *Estado del arte y tendencias en TDD.* Buenos Aires : Universidad Nacional de La Plata, 2011.

117. **Tutorials Point.** *Behavior Driven Development.* s.l. : Tutorials Point, 2016.

118. **Ken Schwaber y Jeff Sutherland.** *The Scrum Guide.* 2013.

119. **Myers, Glenford J.** *The Art of Software Testing.* New Jersey : John Wiley & Sons, Inc.,, 2004. 0-471-46912-2.

120. **Pressman, Roger S.** *SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH.* New York : McGrawHill, 2010. 978–0–07–337597–7.

121. **TOROI, TANJA.** *Testing Component-Based Systems.* Kuopio : Kuopio University Library, 2009. 978-951-27-0113-1.

122. **Trigas Gallego, Manuel .** *Gestión de Proyectos Informáticos: Metodología SCRUM.* 2012.

7. ANEXOS

I. Artículo enviado a las XXI Jornadas de Ingeniería de Software y Bases de Datos (JISBD2017). Universidad de La Laguna. España.

A continuación, se presenta el artículo corto enviado a las JISBD2017. Este artículo fue rechazado, no obstante, consideramos que al ser un trabajo realizado durante la realización de este TFG, y que resume sus objetivos, hemos considerado de interés incluirlo como anexo.

Agile Testing: Principios y Prácticas

Agile Testing: Principios y Técnicas[11]

A.Félix Sánchez-Oro, Amparo Navasa

Quercus Software Engineering Group,
Departamento de Lenguajes y Sistemas Informáticos, Universidad de Extremadura, España.
asanchezld@alumnos.unex.es, amparonm@unex.es

Resumen. Entorno al mundo que rodea a la calidad del software ha surgido recientemente una tendencia denominada *Agile testing*, que consiste en aplicar los principios del manifiesto ágil a las pruebas de software. Estos principios teóricos se materializan en prácticas que se aplican a las fases de prueba de los ciclos de vida ágiles. En este trabajo se muestran distintas técnicas utilizadas para llevar a cabo pruebas en el entorno ágil, así como una comparativa entre la aplicación de pruebas tradicionales y ágiles.

Palabras Clave: *Agile testing*, Pruebas Ágiles, Calidad del Software, Desarrollo Dirigido por Comportamiento, Desarrollo Dirigido por Test, Desarrollo Dirigido por Test de Aceptación.

1 Introducción

En los últimos años ha surgido una nueva tendencia con las pruebas de software. Al igual que surgió una filosofía Ágil para el desarrollo de software ¿por qué no seguir la misma filosofía para las pruebas? Con este concepto aparece un nuevo término asociado al mundo del software, el *Agile Testing* o Pruebas Ágiles.

El *Agile Testing* es una práctica que se incluye en el conjunto de las pruebas de software, que sigue los principios del desarrollo ágil, involucrando en el proceso de las pruebas no solamente al equipo de desarrolladores, sino también poniendo en contacto las pruebas con los líderes del proyecto y con el propio cliente.

Dentro de las pruebas ágiles hay unos principios teóricos que marcan el objetivo final de las pruebas y unas prácticas que materializan estos principios. La aplicación de estas prácticas ágiles se realiza con una metodología ágil como base, generalmente XP (eXtreme Programming) o Scrum. En la sección 2 se comenta muy brevemente los principios de las pruebas agiles; en la sección 3, las técnicas más utilizadas en este entorno y en la sección 4 se comenta el estudio realizado, que compara la tasa de eficacia entre la aplicación técnicas de pruebas agiles y las tradicionales.

[11]Trabajo parcialmente subvencionado por el ProyectoTIN2015-69957-12 (MINECO/FEDER,UE)

2 Principios del *Agile Testing*

El primer principio fundamental, es comprender que *el agile testing no es una fase separada del desarrollo* (103), sino que está integrada en todas ellas y es una actividad obligatoria y tan importante como las demás. La manera de proceder es: *pensar en implementar los test antes de escribir código, como se ve más adelante*.

Por otra parte, la filosofía ágil requiere que *todo el equipo se involucre en el objetivo final*. Si aplicamos esta norma a las pruebas obtenemos un cambio en las competencias de los roles: los test no sólo solucionan los fallos, ahora *representan expectativas de calidad*. Además, no sólo los *testers* son los encargados de ejecutar pruebas; el *agile testing* involucra a desarrolladores y programadores.

El principio ágil de *no generar una documentación muy extensa* y compleja se aplica también al *testing*, al emplear prácticas como el uso de estilos de documentación y herramientas agiles, utilización de test automáticos, aprovechar documentos para varios propósitos o el uso de listas de comprobación.

Aplicar estos principios proporciona características como la *limpieza del código*: se prueba y se recibe *feedback* continuamente. Esta reducción del tiempo de *feedback* se debe a las características del *Agile Testing*. Las pruebas no se hacen al final del sprint, sino que son continuas. Además, se logra un aumento de la calidad del código final.

3 Técnicas del *Agile Testing*

Los principios de las pruebas ágiles mencionados son objetivos implícitos y explícitos que hay que aplicar correctamente. Para cumplirlos y materializar estos principios, hay que seguir buenas prácticas a lo largo del desarrollo del proyecto. A continuación, se muestran algunas técnicas que facilitan la aplicación de estas prácticas.

El **Test Driven Development (TDD)** o Desarrollo Dirigido por Test (104) es una técnica de diseño e implementación procedente de la metodología XP. TDD se usa para diseñar e implementar correctamente los requisitos del software. Resuelve preguntas que un programador se hace al comienzo del desarrollo de un proyecto: por dónde empezar, cómo hacerlo, o qué debemos codificar y que no. Para usar TDD el desarrollador tiene que cambiar su método de trabajo. Ahora, en vez de crear los casos de uso y después implementarlos, va a intentar convertir un caso de uso en un ejemplo. Hacer esto con todos los casos de uso lleva a que los ejemplos describan por sí solos la próxima tarea a implementar. El objetivo de esta técnica es minimizar la creación del código implementando sólo lo imprescindible, lo que el usuario ha pedido, reduciendo los errores y creando software con fácil adaptación a los cambios (siguiendo la filosofía ágil). TDD se basa en tres fases:

1. Se crea la especificación del requisito, es decir, se crea el ejemplo o el test. Esta es la especificación de un caso de uso. La primera duda que surge es cómo se crea un test sin tener el código implementado. La clave está en imaginar la forma del código que se implementará y cómo probar que funciona correctamente. Al principio no se tienen test completos, sino ejemplos o especificaciones. Cada una de ellas individualmente pasará por las demás fases del TDD. Esto se hace así, porque diseñar la especificación y las pruebas antes de llegar a implementarla hace que el código sea radicalmente diferente.
2. Hecha la especificación del requisito o el test, se implementa el código del ejemplo que la hará funcionar. Sólo hay que implementar lo imprescindible para realizar la prueba, sin importar la calidad o limpieza del código. El objetivo es implementar la funcionalidad para que el test se ejecute correctamente. En esta fase al desarrollador le pueden surgir dudas sobre cómo interactuará el futuro flujo de información de la aplicación: bucles, condiciones, entrada y salida de datos etc. La manera de seguir es anotando estas dudas, que serán usadas en sprint futuros.
3. Refactorización del código: el programador busca en el código, líneas y funciones duplicadas o errores lógicos. Hay que revisar no solamente el código de la prueba, sino también el del ejemplo.

El principal problema de TDD se produce cuando un ejemplo pasa correctamente el test y no consigue la funcionalidad deseada. Sólo se puede ver si lo desarrollado se ha codificado correctamente. Pero este es un enfoque miope, sólo centrado en desarrollar código de calidad para ese ejemplo y no para la aplicación en su conjunto.

El **Behaviour Driven Development (BDD)** o Desarrollo Basado en Comportamiento (105) está basado en la técnica anterior, y corrige las deficiencias de ésta. BDD trabaja con historias de usuario en lenguaje natural, y a la vez en código fuente. Las historias de usuario (User Story) (104) es una técnica que facilita la extracción de características o funcionalidades que debe de tener el software. Los

desarrolladores y el cliente escriben estas historias de usuario para extraer los casos de uso que quiere el cliente. El objetivo de BDD es que las historias de usuario dirijan el desarrollo del proyecto software. Además, BDD permite comprobar que el software implementado cumple con la funcionalidad que se esperaba, corrigiendo así la deficiencia de TDD. En BDD, **Gherkin** (98) es el lenguaje compartido entre programadores, cliente y el ordenador. Utiliza las historias de usuario para desarrollar el código fuente, crear pruebas automatizadas, en la mayoría de los casos pruebas unitarias, y para generar documentación interna. Para escribir las historias de usuario con Gherkin, lo primero que hay que definir es la característica o caso de uso mediante las historias de usuario. Una vez definida, se generan los pasos necesarios para que la característica acabe con éxito. Los pasos constan de precondiciones y acciones que llevan a un resultado. Cuando aplicamos este método en varios sprint, se obtiene software estructurado y ordenado.

El Desarrollo Dirigido por Test de Aceptación o **Acceptance Test Driven Development (ATDD)** (106), es otra técnica de diseño similar a TDD. La idea es: se parte de un desarrollo al que los programadores han aplicado TDD. Si lo han hecho correctamente, deberían haber obtenido un código limpio y refactorizado, que ha sido probado por pruebas unitarias. El código, por lo tanto, es bueno, pero ¿sucede lo mismo con la aplicación? ATDD realmente es una metodología de trabajo, que busca comprobar si los pasos que se siguen en el desarrollo son los mejores para llegar al objetivo. Al igual que en TDD, ATDD consta de una serie de pasos secuenciales, pero en ATDD no basta con crear ejemplos y sus test. En esta técnica desarrollador y cliente crean una historia de usuario; con ella deben proponer, crear y aceptar los criterios de aceptación, con ejemplos, para luego ejecutar estas pruebas de aceptación, antes de comenzar el desarrollo del proyecto y, por lo tanto, antes de cualquier requisito del software. Pasos a seguir:

1. Se hace una reunión entre los miembros del equipo donde se aborda qué funcionalidad desarrollar primero y con ella qué historia de usuario crear.
2. Se escriben las pruebas de aceptación, en lenguaje natural. También se crean los ejemplos. En etapas intermedias del proyecto, esta fase sirve para refinar las pruebas de aceptación o crear otras nuevas.
3. El tercer paso se divide en dos etapas. En primer lugar, se implementa el requisito, la prueba, automatizando los ejemplos creados anteriormente, que se quieren transformar en pruebas. Después se aplica TDD al código implementado para obtener un código limpio y sin errores.
4. Por último, se hace una demostración al cliente al final del sprint con las funcionalidades creadas.

4 Comparativa de la eficacia entre pruebas tradicionales y ágiles

El objetivo de esta comparativa es comprobar el grado de eficacia de las pruebas ágiles frente a los métodos de prueba tradicionales en el entorno empresarial. Para ello hemos i) obtenido la información de primera mano en organizaciones españolas y ii) realizado un estudio de bibliografía sobre casos reales en organizaciones que aplican *agile testing* (107). En el primer caso hemos contactado con empresas nacionales que aplican pruebas ágiles y tradicionales para conocer el esfuerzo en la ejecución de las pruebas y los resultados obtenidos. Los resultados preliminares del estudio nos permiten concluir que la aplicación de pruebas ágiles tiene unas tasas de éxito mayores que la aplicación de pruebas tradicionales, si bien, esto supone un incremento del coste del desarrollo. También hemos comprobado que el *agile testing* no funciona igual de bien en todos los proyectos. Hay desarrollos que son susceptibles a la aplicación de pruebas ágiles, pero hay otros tipos de proyectos en los que la aplicación del *agile testing* no funciona igual de bien.

Referencias

1. M. M. Martin C. Robert, Agile Principles, Patterns, and Practices in C#, Prentice Hall, 2006.
2. C. B. Jurado, Diseño Ágil con TDD, safeCreative, 2010.
3. D. Chelimsky, The RSpec Book: Behaviour-Driven Development, Dallas, Texas, USA: The Pragmatic Bookshelf, 2012.
4. M. Wynne, The Cucumber Book, Pragmatic Programmers, LLC, 2012.
5. M. Gartner, ATDD by Example: A Practical Guide to Acceptance, Westford, Massachusetts, USA: Pearson Education, Inc., 2013.
6. K. M. M. Ahsan Nawaz, SOFTWARE TESTING PROCESS IN AGILE DEVELOPMENT, Ronneby, Sweden: Department of Interaction and System Design Blekinge Institute of Technology.

II. MODELO DE ENTREVISTA

En este anexo se presenta el formulario de entrevista que se utilizó durante las entrevistas personales para recopilar los datos.

	UNIVERSIDAD DE EXTREMADURA **ESCUELA POLITÉCNICA** ***ENTREVISTA***	ESCUELA POLITÉCNICA

Entrevistas a los profesionales de las empresas

Información de los entrevistados

Nombre	
Correo electrónico	
Años en la empresa	
Cargo	
Fecha	

Observaciones: **Comentarios:** **Resumen:**

Información general

Descripción

Esta entrevista está diseñada para obtener información acerca del método de trabajo de la empresa, su experiencia y su productividad con pruebas ágiles y tradicionales sobre sus desarrollos. Para ello se procede de la siguiente manera:

1. Agradecimiento por la participación en este estudio
2. Se expone el motivo de la entrevista y se detalla el contenido del estudio
3. Se describe el funcionamiento de la entrevista: consentimiento, duración, lugar y grabación en medios digitales.

Entrevista

1. **¿A qué se dedica en la empresa?**
2. **¿Qué tipos de proyectos realizan?**
3. **¿Los desarrollos son completos o son productos que luego se exportan (SDK) a otros desarrolladores?**
4. **¿Cuántas personas suelen participar en un proyecto?**
5. **¿Cómo es la jerarquización de los grupos de los proyectos?**
6. **¿Qué tipo de ciclo de vida o desarrollo utilizáis?**
7. **Respecto a las pruebas ¿utilizáis un equipo de testing distinto de los programadores? ¿o bien son los propios programadores los que prueban el código?**
8. **¿Las pruebas son una fase más del desarrollo o están integradas en otras fases?**
9. **¿Cuál es vuestra filosofía de pruebas de software?**
10. **¿Cuándo es la primera vez que se habla acerca de las pruebas en el proyecto?**
11. **¿Hacéis alguna práctica de *testing agile* o son pruebas tradicionales?**
12. **¿Hay algún método concreto para hacerlas?**
13. **¿Habéis cambiando el método de pruebas en estos últimos 5 años?**
14. **¿Tienen las pruebas ágiles algún incremento en el presupuesto respecto a desarrollo anteriores con pruebas tradicionales?**

III. MODELO DE CONSENTIMIENTO INFORMADO

Este anexo es un documento que se entregaba al entrevistado para que tuviera pleno control (derechos ARCO) sobre la información que nos cedía y que será divulgada en este TFG.

	UNIVERSIDAD DE EXTREMADURA **ESCUELA POLITÉCNICA** ***CONSENTIMIENTO INFORMADO***	ESCUELA POLITÉCNICA

Yo _____

mayor de edad, con número de DNI_____ autorizo a Antonio Félix Sánchez-Oro Portillo, estudiante del Grado en Ingeniería Informática en Ingeniería de Software, a utilizar y/o publicar total o parcialmente este formulario/entrevista, con fines científicos, educativos o culturales.

Cáceres a ___ de _____ del 2017.

Fdo.: _____

Colaborador del TFG

Los datos de carácter personal facilitados serán incorporados al TFG "Testing Agile", titularidad de la Universidad de Extremadura, con la finalidad de llevar a cabo un

estudio comparativo y elaborar así, un Trabajo de Fin de Grado del Grado de Ingeniería Informática en Ingeniería de Software. Es necesario tener presente que vuestros datos de carácter personal estarán correctamente disociados si se publica este Trabajo de Fin de Grado en el Depósito de la Universidad de Extremadura. El órgano responsable del fichero es la Secretaría General. En cualquier caso, podéis acceder a los derechos de acceso, rectificación y cancelación mediante una comunicación escrita, adjuntando fotocopia del DNI u otro documento identificativo, dirigida al Rectorado de Cáceres con dirección: Plaza de Caldereros, 1, 10003 Cáceres. Teléfono: 927-257005. Fax: 927-257028. E-mail: secgral@unex.es

IV. MODELO DE FORMULARIO

Este anexo es un formulario de apoyo a la entrevista. Es un documento utilizado por el entrevistador (el autor de este TFG) para facilitar las labores de obtención de información. También se utilizó este formulario para hacer la encuesta online y así poder recopilar más datos.

	UNIVERSIDAD DE EXTREMADURA **ESCUELA POLITÉCNICA** ***FORMULARIO***	ESCUELA POLITÉCNICA

Información acerca del trabajador

Departamento:

- ☐ Programador
- ☐ Tester
- ☐ Jefe de equipo
- ☐ Jefe de desarrollo
- ☐ Miembro del equipo
- ☐ Otro
- ☐ Director o superior

Duración del puesto en la empresa:

- ☐ Menos de 1 año
- ☐ Entre 1 y 2 años
- ☐ Entre 3 y 4 años
- ☐ Entre 5 y 9 años
- ☐ Entre 10 y 19 años
- ☐ 20 años o más

Información acerca de la empresa

Tamaño de la empresa:

- ☐ 5 trabajadores o menos
- ☐ Entre 6 y 9
- ☐ Entre 10 y 19
- ☐ Entre 20 y 49
- ☐ Entre 50 y 99
- ☐ Más de 100

Proyectos o líneas de negocio simultáneas:

- ☐ 1 Proyecto
- ☐ 2 o 3 proyectos
- ☐ Entre 4 y 6 proyectos
- ☐ Entre 7 y 10 proyectos
- ☐ Más de 11 proyectos

Personal asignado por proyectos:

- ☐ 5 trabajadores o menos
- ☐ Entre 6 y 9
- ☐ Entre 10 y 19
- ☐ Entre 20 y 49
- ☐ Entre 50 y 99
- ☐ Más de 100

Ciclos de vida que alguna vez ha usado la empresa:

- ☐ Cascada (Waterfall)
- ☐ Espiral
- ☐ Iterativo e Incremental
- ☐ Ágil
- ☐ Orientado a Reutilización
- ☐ Otro

Ciclos de vida actual de la empresa:

- ☐ Cascada (Waterfall)
- ☐ Espiral
- ☐ Iterativo e Incremental
- ☐ Ágil
- ☐ Orientado a Reutilización
- ☐ Otro

Tipo de metodología ágil:

- ☐ Scrum
- ☐ Scrum/XP hibrido
- ☐ Personalizado
- ☐ XP (Xtreme Programming)
- ☐ MSF (Micosoft Solution Framework)
- ☐ DSDM (Dynamic Systems Development Method)
- ☐ OSS (Open Source Software Development)
- ☐ Otro
- ☐ Scrumban
- ☐ Kanban
- ☐ FDD (Feature Driven-Development)
- ☐ RUP, AUP y EUP (Rational/Agile/Enterprise Unified Process)
- ☐ LD (Lean Development)
- ☐ ASD (Adaptative Software Development)
- ☐ Crystal Methodologies

Tasa de éxito actual por proyectos

- ☐ 100%
- ☐ Entre un 75% y 100%
- ☐ Entre un 50% y 75%
- ☐ Entre un 25% y un 50%
- ☐ Inferior a un 25%

Tasa de éxito anterior a la metodología actual (si la hubiera)

- ☐ 100%
- ☐ Entre un 75% y 100%
- ☐ Entre un 50% y 75%
- ☐ Entre un 25% y un 50%
- ☐ Inferior a un 25%

Información sobre las pruebas de software

Fase en la que se programan las pruebas

- ☐ Pruebas tradicionales (una fase más del desarrollo)
- ☐ Pruebas ágiles (integradas en todas las fases del desarrollo)

Prácticas ágiles utilizadas

- ☐ TDD
- ☐ BDD
- ☐ ATDD
- ☐ Específica de la empresa
- ☐ Otro

Método de realización para las pruebas

- ☐ Existe un equipo de testing externo a los programadores.
- ☐ Los programadores son los únicos encargados de elaborar y ejecutar las pruebas de forma colectiva. (Testing Colaborativo)
- ☐ El método de desarrollar software es ágil (TDD o ATDD)
- ☐ Los programadores son los únicos encargados de elaborar y ejecutar las pruebas de forma individual.
- ☐ Los programadores tienen ayuda de otros miembros del equipo
- ☐ El método de desarrollar software es ágil (BDD o BDD + TDD)

Fase en la que se programan las pruebas

- ☐ Definición y Análisis
- ☐ Implementación
- ☐ Diseño
- ☐ Otro

Tiempo dedicado por las pruebas

- ☐ Entre el 10% y el 20% del tiempo para el proyecto
- ☐ Entre el 20% y el 40% del tiempo para el proyecto
- ☐ Entre el 40% y el 60% del tiempo para el proyecto
- ☐ Entre el 60% y el 80% del tiempo para el proyecto

Impacto económico de las labores de pruebas en el proyecto

- ☐ 100%
- ☐ Entre un 25% y un 50%
- ☐ Entre un 75% y 100%
- ☐ Entre un 25% y un 10%
- ☐ Entre un 50% y 75%
- ☐ Inferior a un 10%

Diferencia entre aplicar pruebas ágiles y tradicionales en eficacia

- ☐ Usar testing ágil es más eficaz siempre
- ☐ Usar testing ágil es más eficaz en la mayoría de los casos
- ☐ Al usar testing ágil no hemos notado diferencias de eficacia
- ☐ Usar testing ágil no es eficaz en la mayoría de los casos
- ☐ Usar testing ágil no es eficaz en ningún caso

Diferencia entre aplicar pruebas ágiles y tradicionales en tiempo

- ☐ Usar testing ágil ahorra tiempo siempre
- ☐ Usar testing ágil ahorra tiempo en la mayoría de los casos
- ☐ Al usar testing ágil no hemos notado diferencias de tiempo
- ☐ Usar testing ágil no ahorra tiempo en la mayoría de los casos
- ☐ Usar testing ágil no ahorra tiempo en ningún caso
- ☐ Usar testing ágil provoca una pérdida de tiempo

Diferencia entre aplicar pruebas ágiles y tradicionales en el presupuesto

- ☐ Usar testing ágil es más barato siempre
- ☐ Usar testing ágil es más barato en la mayoría de los casos
- ☐ Al usar testing ágil no hemos notado diferencias en el presupuesto
- ☐ Usar testing ágil ahorra dinero en la mayoría de los casos
- ☐ Usar testing ágil no ahorra dinero en ningún caso
- ☐ Usar testing ágil incrementa el presupuesto

Comentarios

¿Echa algo en falta en su entorno de trabajo que pudiera ayudarle a mejorar el rendimiento?

Gracias por su colaboración.