10. Pruebas, Calidad y la Mejora del Proceso de Desarrollo de Software

10.1. Introducción y objetivos

Al comienzo del tema se realizará un estudio de los conceptos de verificación y validación del *software* y el tipo de pruebas que abarcan. Junto a esto, se estudiará una de las técnicas más populares y comúnmente empleadas para guiar el proceso de desarrollo de *software:* *Test Driven Development (*TDD) y algunas de sus extensiones, como ATDD (*Acceptance Test Driven Development*) o BDD (*Behavior Driven Development*).

Además, se estudiará el concepto de calidad del *software,* que es más complejo de lo que puede parecer a primera vista. Se trata de un concepto abstracto, cuya presencia es difícil de definir, pero su ausencia es fácilmente detectable. Para ayudar

a determinar el conjunto de características y parámetros, que es necesario evaluar desde diferentes puntos de vista, hay diferentes marcos y modelos. En este tema se analizan el conjunto de normas de la familia ISO/IEC 25000, que pretenden ayudar y guiar en este sentido.

Por último, en este tema también se estudiará la mejora del proceso de *software* y los modelos de madurez que pueden ayudar a las organizaciones a sentar las bases para la optimización de sus procesos. Entre ellos destaca el modelo de madurez de capacidades integrado.

Con el estudio de este tema se pretende alcanzar los siguientes objetivos:

* Diferenciar claramente los conceptos de verificación y validación del software, y su importancia en el proceso de desarrollo.
* Saber diferenciar los diferentes tipos de pruebas de software que se realizan durante el desarrollo de un sistema de software.
* Comprender los principios fundamentales de la técnica TDD y su importancia y beneficios en el proceso de desarrollo de software.
* Comprender las características de la técnica ATDD y las fases que comprende.
* Conocer la técnica de desarrollo BDD.
* Analizar el concepto de «calidad de software» y los elementos que la condicionan.
* Analizar el conjunto de normas de la familia ISO/IEC 25000 y sus elementos principales.
* Comprender la importancia de la mejora del proceso de *software* para aumentar la eficiencia de los procesos y mejorar la calidad del producto final.
* Conocer los aspectos generales del modelo CMMI para la mejora de procesos.

10.2. Verificación y validación del software

La verificación y validación del *software* abarca el conjunto de procesos de comprobación y de análisis que permiten asegurar que un producto es acorde con su especificación y cumple con las necesidades para las que fue construido. Sin embargo, ambas palabras representan enfoques diferentes en este proceso:

* La verificación comprueba que el software se construye de la manera correcta. Comprueba, en definitiva, el correcto funcionamiento del *software* y que el producto construido está libre de errores.

Responde a la pregunta: «¿estamos construyendo correctamente el sistema?».

Se centra en relacionar el producto con el conjunto de requisitos funcionales y no funcionales de la especificación, y los productos de cada fase con las especificaciones obtenidas al final de la anterior.

* La validación confirma que el producto construido es apropiado para el uso previsto.

En otras palabras, responde a la pregunta: «¿estamos construyendo el sistema correcto?».

En este caso se contrasta el producto con las expectativas del cliente o usuario.

Estos conceptos aparecen reflejados en la Figura 1.

A diagram of a system

Description automatically generated

Figura 1. Diferencia entre los conceptos de verificación y validación del software. Fuente: elaboración propia.

Mientras que las tareas de verificación se centran en analizar el propio proceso de desarrollo, comprobando que cada paso es realizado correctamente y en conformidad con los pasos anteriores, la validación se centra en garantizar que el sistema en construcción es compatible en todo momento con las expectativas de los clientes o usuarios finales, y cumplirá finalmente con su misión.

Aunque la verificación determina la calidad del producto, desde el punto de vista de la ingeniería, no siempre garantiza que el producto sea útil. Los procesos de verificación están formados por actividades bastante objetivas, donde tiene poca influencia la posible subjetividad del usuario final. En ellos se analizan minuciosamente los productos parciales de trabajo (métodos, clases, módulos y su interacción, documentos de diseño y documentación del producto), pero todo ello no garantiza que la validación sea exitosa, pues este paso final depende precisamente de las expectativas del usuario que, finalmente, trabajará con la aplicación.

En la Tabla 1 se resumen las principales características y actividades de estos dos procesos complementarios entre sí.

A table with text on it

Description automatically generated

Tabla 1. Comparativa entre los procesos de verificación y validación del software. Fuente: adaptada a partir de Sharma, 2021.

A pesar de esta definición de actividades orientativas que se recogen en la Tabla 1, lo cierto es que, según Pressman (2010), hay una «fuerte divergencia de opinión acerca de qué tipos de pruebas constituyen la validación» (p. 384):

* Algunos autores opinan que todas las pruebas de *software* sirven para la verificación (es decir, para comprobar que el sistema se construye correctamente), mientras que la validación se lleva a cabo en dos momentos:
  + Cuando los requisitos se revisan (asegurando que la especificación refleja los deseos del usuario).
  + Cuando el usuario evalúa el sistema en funcionamiento y lo acepta, y en este sentido equivaldría a las pruebas de aceptación.
* Otros autores contemplan las pruebas unitarias y de integración más relacionadas con la verificación, mientras que las de orden superior (como las pruebas de validación y las pruebas del sistema) tienen que ver con la validación del sistema.

10.3. Las pruebas de software

Podemos definir las pruebas de *software, s*egún Bolaños, Sierra y Alarcón (2007), como «el proceso que ayuda a identificar la corrección, completitud, seguridad y calidad del software desarrollado» (p. 1).

Las pruebas consisten en ejecutar un conjunto de elementos *software* con el objetivo de encontrar errores. Por tanto, las pruebas no pueden garantizar que el *software* esté completamente libre de errores ni, tan siquiera, que el programa sea correcto, pero sí intentan cubrir, de una manera amplia, posibles errores, con el fin de detectarlos en caso de que se produzcan.

Podemos identificar diferentes estrategias para realizar las pruebas de un sistema, que se ejecutan de dentro hacia fuera: comenzando por pequeños fragmentos de código y módulos unitarios, y avanzando durante la integración hasta finalizar ejecutando pruebas sobre el sistema completo. Aquí veremos las siguientes estrategias (Bolaños *et al.*, 2007; Pressman, 2010):

* Pruebas unitarias: se encargan de comprobar la correcta lógica y funcionalidad de cada elemento aislado (método, clase o módulo), teniendo en cuenta su especificación.
* Pruebas de integración: comprueban el correcto funcionamiento de dos o más elementos integrados, y el flujo de información a través de las interfaces que exponen.
* Pruebas de validación: comprueban la compatibilidad del sistema respecto de los requisitos de usuario y del sistema.
* Pruebas de sistema: comprueban el correcto funcionamiento del sistema en un entorno operativo de producción, prestando especial atención a los requisitos no funcionales.
* Pruebas de aceptación: en este caso es el usuario o cliente quien valida que el producto se ajusta a sus necesidades.

En un enfoque tradicional, estas pruebas se corresponden con momentos diferentes del ciclo de vida del desarrollo, como se muestra en la Figura 2.

A diagram of a diagram

Description automatically generated

Figura 2. Modelo de proceso de *software* en V. Fuente: elaboración propia.

El modelo de la Figura 2, modelo de proceso de *software* en V, donde se muestran las relaciones entre cada fase del ciclo de desarrollo del producto y la fase de pruebas asociada, también es conocido como «modelo de verificación y validación».

De todas las pruebas anteriores, las pruebas unitarias, de integración y del sistema son realizadas únicamente por el equipo de desarrollo, mientras que en las de validación y aceptación participa, además, el cliente o usuario final del producto.

Las pruebas unitarias

Las pruebas unitarias se centran en una unidad de código claramente diferenciada del resto, generalmente una función o método de un objeto.

Cuanto mejor diseñadas estén y más amplitud de aspectos cubran sobre la funcionalidad diseñada, más seguro será para el programador introducir cambios en el código.

Idealmente, las pruebas unitarias deben estar aisladas de otras dependencias. Si un fragmento de código debe acceder a la red o realizar operaciones frente a la base de datos, existen paquetes informáticos que permiten simular estas dependencias de manera controlada (Hartikainen, 2015).

Como vemos en la Figura 3 las pruebas unitarias son un elemento independiente del código que se está verificando, pero que facilita su ejecución.

En la mayoría de las ocasiones, el componente bajo prueba no es un programa ejecutable y, por tanto, hay que desarrollar un *software* controlador o «programa principal» que acepta datos de los casos de prueba, los envía al componente y genera un informe de resultados.

En la Figura 3 los representantes sustituyen otros módulos de los cuales depende el componente probado, simulando su comportamiento en el contexto de las pruebas unitarias.

A diagram of a computer network

Description automatically generated

Figura 3. Las pruebas unitarias ejecutándose sobre un código en producción. Fuente: Fowler, 2014; Pressman, 2010.

Las pruebas de integración

En este caso, se trata de comprobar el funcionamiento conjunto de diferentes partes del código (por ejemplo, comprobar que los accesos a una base de datos funcionan correctamente cuando se ejecutan frente a una base de datos real).

Mientras que durante las pruebas unitarias estas interacciones se ejecutan de manera simulada, aquí interactúan los subsistemas desarrollados entre sí o con otros previamente existentes. En este caso, lo habitual es realizar la integración de manera progresiva o incremental, y no hacerla con todos los módulos a la vez (Bolaños *et al.*, 2007, p. 21; Pressman, 2010, p. 392).

Las pruebas funcionales o pruebas de validación

Estas pruebas verifican que se cumplen los requisitos funcionales, que son traducción directa de las funcionalidades esperadas por el usuario a un dominio técnico.

Se realizan sobre el sistema integrado y se centran en acciones visibles para el usuario, y en las salidas del sistema que este reconoce. La validación del *software* se consigue aquí, según la conformidad del producto con los requisitos funcionales, y se realiza siguiendo un plan de pruebas previamente definido. En esta fase es fundamental la participación del cliente o usuario final, y la comprobación de que los criterios de validación definidos en la especificación inicial se verifican.

Algunos autores, como Roger Pressman, no diferencian entre las pruebas de validación y las pruebas de aceptación, que veremos más adelante, siendo todas ellas de tipo funcional (Pressman, 2010).

Podemos destacar dos tipos de pruebas específicas en esta fase, especialmente importante cuando se realiza un *software* a medida para un cliente:

* Pruebas alfa
  + Son llevadas a cabo en el sitio del desarrollador.
  + Participa un grupo representativo de los usuarios finales, que puede ser un colectivo demasiado amplio.
  + Se realizan en un ambiente controlado.
  + El desarrollador toma nota de los problemas encontrados para corregirlos.
* Pruebas beta
  + Realizadas en uno o más sitios de usuarios finales.
  + El desarrollador no está presente.
  + La ejecución se realiza en un ambiente de uso real, donde el cliente registra los problemas detectados para su corrección.
  + Cuando durante las pruebas beta el cliente somete el producto a un conjunto de pruebas específicas, con el objetivo de descubrir errores antes de la finalización del proyecto, estas pruebas coinciden con las pruebas de aceptación.

Pruebas del sistema

No debemos olvidar que cualquier desarrollo *software* generalmente se enmarca en el contexto de un sistema de información más amplio; bien porque el sistema debe interactuar con otros sistemas de información o bien porque su despliegue va a producirse en un ambiente *software* y *hardware* diferente al de desarrollo.

Las pruebas del sistema persiguen ejercitar el conjunto del sistema de información, comprobando la correcta interacción de todos los componentes en un ambiente de producción.

En este contexto, adquieren especial importancia los requisitos no funcionales que, como sabemos, generalmente afectan al sistema en su conjunto e imponen restricciones que perjudican al sistema en funcionamiento en un entorno operativo real.

En ocasiones, estas pruebas se realizan antes que las pruebas de validación (especialmente cuando la validación coincide con la aceptación) o, incluso, de manera simultánea (Bolaños *et al.*, 2007).

Algunas pruebas específicas importantes en esta etapa son las siguientes:

* Pruebas de recuperación, que comprueban en qué medida el sistema es capaz de recuperar su propio estado y los datos con los que trabaja ante un fallo, reanudando el normal funcionamiento.
* Pruebas de seguridad, que intentan garantizar que el sistema cuenta con mecanismos de protección adecuados frente a intentos de penetración, intencionados o accidentales, que busquen entorpecer el funcionamiento del sistema o robar información.
* Pruebas de esfuerzo, que ponen al sistema en un modo de funcionamiento anómalo en el que se demandan recursos en una cantidad, frecuencia o volumen anómalos, intentando responder a la pregunta «¿cuánto podemos doblar esto antes de que se rompa?» (Pressman, 2010, p. 402).
* Pruebas de rendimiento, que evalúan el comportamiento del sistema en condiciones normales de operación y, generalmente, se emparejan con las pruebas de esfuerzo.
* Pruebas de despliegue, que son apropiadas cuando el sistema debe ejecutarse en plataformas diferentes con entornos de *hardware* o de *software* distintos. Estas pruebas también son conocidas como pruebas de configuración, pues verifican el correcto funcionamiento del producto en diferentes configuraciones del entorno operativo.

Pensemos, por ejemplo, en una aplicación web que debe funcionar del mismo modo en diferentes navegadores, o en diferentes combinaciones de navegador web y sistema operativo.

* Pruebas de documentación, en las que se comprueba la adecuación de la documentación de usuario a su propósito.

Las pruebas de aceptación

Se centran en comprobar de manera automática el correcto funcionamiento de una aplicación entera.

Para ello, es útil utilizar herramientas como «selenium», en el caso de aplicaciones web, que permiten simular la interacción con un navegador web.

Las pruebas de aceptación representan los intereses del cliente, y le aportan la seguridad de que el *software* desarrollado incluye todas las características deseadas y se comporta adecuadamente.

El hecho de que todas las pruebas de aceptación se ejecuten de manera correcta es un hito fundamental en el desarrollo de un proyecto, pues quiere decir que el proyecto ha finalizado y que cumple con todas las especificaciones del cliente.

A table with text on it

Description automatically generated

Tabla 2. Comparativa entre las pruebas unitarias y las pruebas de aceptación. Fuente: Gal-Oz y Balaban, 2016.

10.4. Test Driven Development (TDD)

Podemos considerar las pruebas de *software* como la actividad que más ayuda a garantizar la buena calidad del código. Sin embargo, paradójicamente, es una actividad que tradicionalmente se venía realizando en las últimas fases de los proyectos, cuando es conocido que el coste de realizar cualquier cambio resulta mucho más alto, como vemos en la Figura 4.

Parece, por tanto, que si conseguimos adelantar esta fase hacia la izquierda, tendremos la oportunidad de reducir posibles costes adicionales ocasionados por la detección tardía de defectos.

A diagram of a software development process

Description automatically generated

Figura 4. La tradicional curva del «coste del cambio» con el modelo de proceso en cascada en el fondo. Fuente: Murphy, 2005.

Los nuevos enfoques de desarrollo ágil consiguen esto de alguna manera, solapando todas las actividades dentro de cada iteración, de manera que en cada una de ellas se incluye un poco de análisis de requisitos, un poco de diseño, un poco de implementación y las pruebas asociadas, hasta que el sistema está completo.

El desarrollo dirigido por pruebas o TDD (*Test Driven Development*) es un estilo de programación en el que se entremezclan tres actividades (Agile Alliance, 2015a):

* Codificación.
* Pruebas (en forma de pruebas unitarias).
* Diseño (en la forma de refactorización del código).

El TDD alcanza pleno sentido dentro de los modelos de desarrollo iterativos, orientados a tener en consideración posibles cambios inesperados en los requisitos.

El enfoque de desarrollo con las «pruebas primero» (*test-first*)

Podemos considerar el desarrollo dirigido por pruebas como una técnica de construcción de *software* que guía el proceso de desarrollo comenzando por la escritura de las pruebas (Fowler, 2005).

Son las pruebas las que guían el desarrollo de una clase o un fragmento de código. También se conoce como enfoque *test-first* («las pruebas primero»), siendo este uno de los principios fundamentales de la técnica (Murphy, 2005).

*Test-first* ha sido desarrollada fundamentalmente en el contexto de la técnica *eXtreme Programming*, de Kent Beck, y esencialmente consiste en repetir de manera iterativa la siguiente secuencia de tres pasos:

1. Escribir una prueba que verifique la implementación de la siguiente funcionalidad que se va a incluir en el software. En un primer momento la prueba no funcionará y es posible que ni siquiera compile el código por faltar aún elementos de la implementación. En este momento decimos que los test están «en rojo».
2. Codificar la funcionalidad de manera progresiva e incremental hasta conseguir que la prueba se ejecute sin problemas. Al final de este proceso todos los test deberían estar «en verde».
3. Refactorizar el código nuevo y el antiguo para mejorar su estructura interna. Aquí es posible eliminar o reorganizar el código que se ha generado durante el proceso de codificación, mientras se intentaba conseguir que la prueba se ejecutara correctamente. El desarrollador se obliga a reflexionar sobre el código construido, mejorando iterativamente el diseño.

A diagram of a test

Description automatically generated

Figura 5. Ciclo de trabajo con TDD. Fuente: elaboración propia.

De esta manera, se construye de manera incremental toda la funcionalidad del sistema. Este proceso se puede resumir en uno de los mantras de TDD: *red/green/refactor.*

El objetivo final es conseguir código limpio que funciona (*clean code that works*), aunque primero se consigue que el código funcione correctamente y luego nos centramos en los aspectos de limpieza, organización y correcta estructura interna, siempre comprobando que las pruebas no dejan de funcionar correctamente.

En la Figura 6 se muestra una comparación entre enfoques de desarrollo más tradicionales y el propuesto en TDD.

A diagram of a code

Description automatically generated

Figura 6. Flujo de desarrollo: (a) enfoque tradicional, con las pruebas al final y (b) Test Driven Development. Fuente: Janzen y Saiedian, 2008.

En un proceso tradicional es importante realizar un gran esfuerzo inicial en la especificación de la arquitectura del sistema y el diseño detallado antes de comenzar un desarrollo de código significativo. Así, las pruebas unitarias se definen y ejecutan una vez implementado el código.

Por el contrario, el enfoque TDD puede partir de un diseño arquitectónico general del sistema, pero no se continúa con un diseño detallado, sino que se comienza escribiendo las pruebas unitarias, que guían el proceso de desarrollo en iteraciones generalmente cortas y rápidas. De esta manera, el diseño va emergiendo y evolucionando de manera natural.

Las ventajas de TDD

Este enfoque general ofrece dos ventajas fundamentales (Fowler, 2005):

* Ayuda a construir código que se comprueba automáticamente (*self-testing code*).
* Se mejora el diseño. Fundamentalmente en el contexto de la programación orientada a objetos, además ayuda a reflexionar sobre las interfaces con los objetos. En primer lugar, pensamos en objetos vacíos, sin implementación, cuyo estado podemos comprobar a través de los métodos que exponen. De esta manera estamos separando conceptualmente la interfaz de la implementación.

Estos beneficios también han sido comprobados en diferentes estudios experimentales. El trabajo de Janzen y Saiedian (2008) se centró en entrevistas a representantes de cuatro grandes empresas de desarrollo de software. Lo primero que descubrieron es que muchos desarrolladores tenían dos errores de concepto:

* TDD es lo mismo que pruebas automatizadas. Aunque TDD tiene poco sentido sin contar con algún tipo de herramienta que permita automatizar la ejecución de las pruebas, la mera automatización de este proceso no implica que las pruebas se definan en un primer momento.
* TDD implica escribir todos los test al principio. TDD impulsa la codificación en pequeñas iteraciones incrementales, de manera que en cada momento el foco se centra en una funcionalidad concreta que se analiza, se define la prueba correspondiente y, a partir de ese momento, se construye ese fragmento de funcionalidad.

Las conclusiones de este estudio se pueden resumir en los siguientes puntos y análisis de métricas:

* Cobertura de pruebas. Los equipos TDD alcanzan un mayor porcentaje de cobertura del código con sus pruebas.
* Tamaño del código. La métrica más sencilla para la complejidad del código es su tamaño y, en general, un menor número de líneas indica una mayor simplicidad que, en principio, favorece esta técnica.
* Acoplamiento (*coupling*), que indica el grado de interdependencia entre los diferentes módulos o clases del código, dando una medida de lo fuerte que es la conexión entre los diferentes elementos del código. En general, se busca un bajo acoplamiento como síntoma de un buen diseño y estructura del código, donde se facilita la posible sustitución de ciertos elementos.
* Cohesión (*cohesion*), que se refiere al grado en que los elementos, dentro de un mismo módulo o clases, están relacionados entre sí y tiene sentido mantenerlos juntos, midiendo, por tanto, la fortaleza de las relaciones entre los métodos y datos de una misma unidad de código. En general, se busca una alta cohesión como indicador de una buena calidad del código y su diseño.
* Complejidad. El tamaño del código es una buena medida de la complejidad. En la medida en que las clases y métodos sean pequeños serán más sencillos de comprender y mantener.

TDD como técnica de análisis y diseño

Finalmente, debemos resaltar que, a pesar de su nombre, *Test Driven Development,* esta técnica es, ante todo y fundamentalmente, una práctica orientada a la mejora del diseño del *software.*

Los pioneros de esta técnica lo dejan muy claro (Beck, 2001): «Test-first coding isn’t testing» Beck (p. 87). Efectivamente, las pruebas son importantes (como hemos visto esta técnica alcanza un mayor porcentaje de cobertura del código y, además, tiene poco sentido si las pruebas no están automatizadas) pero lo fundamental es que obliga a realizar un esfuerzo de reflexión por parte del programador que repercute en un mejor diseño del *software.*

Supongamos que estamos desarrollando una calculadora. La calculadora debe poder realizar operaciones de suma de dos números. Bajo el enfoque TDD, lo primero que deberíamos preguntarnos es: «¿cómo podemos comprobar esta funcionalidad?».

Supongamos que, después de pensarlo un tiempo, escribimos la siguiente prueba unitaria:

A screenshot of a computer code

Description automatically generated

Código 1. Ejemplo de prueba unitaria. Fuente: elaboración propia.

Evidentemente, si ejecutamos este código, tendremos muchos errores. La clase calculadora no está aún implementada. Eso será lo primero que tendremos que hacer. Pero, escribiendo este código, hemos decidido ya algunas cosas importantes:

* Que la calculadora podrá sumar valores enteros y devolverá como resultado valores enteros.
* Que el método que realiza la operación de suma es un método estático.
* Hemos evaluado posibles alternativas de valores positivos y negativos que se proporcionan como entrada a la operación de suma.
* Nos hemos asegurado de que la propiedad conmutativa se verifica para un par de valores concretos.

Todo esto se podría mejorar, por ejemplo, aumentando la batería de valores de prueba para cubrir un número mayor de alternativas. A partir de este momento comenzaría la implementación de la clase calculadora hasta que todas las pruebas se ejecutasen satisfactoriamente. Es fundamental el hecho de que la prueba queda asociada al código del programa, de manera que cualquier cambio futuro sobre la implementación de la clase será comprobado para que, al menos, no se degrade la funcionalidad de suma de enteros. Pero más importante aún es lo siguiente:

* Como programadores hemos realizado un esfuerzo de análisis. Hemos decidido qué vamos a programar y qué no (el alcance del código).
* También hemos realizado un esfuerzo de diseño. «TDD es una técnica de diseño lógico» (Beck, 2001, p. 88), puesto que al escribir el test estamos definiendo la lógica del comportamiento de la funcionalidad que vamos a implementar.

El patrón AAA

El patrón AAA, —*arrange act assert* (preparar, actuar/ejecutar, afirmar/comprobar)— es una manera habitual de definir las pruebas unitarias.

* La sección *arrange* se utiliza para inicializar objetos y variables que luego utilizaremos en el método o secuencia de acciones que estamos comprobando.
* La sección *act* invoca el método o código analizado empleando las variables que hemos preparado.
* La sección *assert* verifica los resultados del método, comparándolos con los resultados esperados.

A continuación, podemos ver dos ejemplos de pruebas unitarias siguiendo este patrón; uno de ellos en lenguaje C# y otro en lenguaje Java.

A computer screen shot of a code

Description automatically generated

Código 2. Dos ejemplos de prueba unitaria en C# siguiendo el patrón AAA. Fuente: Microsoft, 2022.

A black background with text on it

Description automatically generated

Código 3. Ejemplo de prueba unitaria en Java siguiendo el patrón AAA. Fuente: Grigg, 2012.

*Frameworks* y herramientas para TDD

Existen múltiples *frameworks* que permiten automatizar las pruebas que realizamos sobre el sistema en construcción:

* Serenity, por ejemplo, permite automatizar las pruebas en Java, integrándose también con herramientas BDD, como Cucumber o JBehave.
* Robot framework es un *framework* para pruebas automatizadas en Python, aunque también es posible utilizarlo en proyectos Java, a través de Jython o en proyectos .NET, mediante IronPython (Pablos, 2018).
* NUnit es un *framework* de *testing* compatible con todos los lenguajes .NET (NUnit, 2022).
* PHPUnit es un *framework* para realizar pruebas unitarias en lenguaje PHP (PHPUnit, 2022).
* JUnit es un *framework* para hacer pruebas unitarias de aplicaciones Java (JUnit, 2022).

Como ejemplo, en los siguientes vídeos aprenderás a implementar un pequeño proyecto.

TDD en Visual Studio (C# y NUnit)

En este ejemplo se usa Visual Studio, el lenguaje C# y NUnit, un *framework* de *testing* compatible con todos los lenguajes .NET (NUnit, 2022).

En el vídeo, *TDD en Visual Studio,* se implementa el juego FizzBuzz, cuya lógica es aparentemente complicada de implementar para muchos programadores, y se utiliza habitualmente tanto en introducciones a la técnica TDD como en entrevistas de trabajo (Ghory, 2007; WikiWikiWeb, 2014).

El enunciado es el siguiente: «Escribe un programa que muestre en orden los números del 1 al 100. Pero cuando el número sea múltiplo de 3, se mostrará la palabra "Fizz" en vez del número, y cuando sea múltiplo de 5, se mostrará la palabra "Buzz" en vez del número. Si el número es múltiplo de 3 y de 5, se mostrará la palabra "FizzBuzz"».

10.5. Acceptance Test Driven Development (ATDD)

El desarrollo dirigido por pruebas de aceptación o ATDD (*Acceptance Test Driven Development*) es análogo al TDD, pero involucra a miembros del equipo que pueden aportar diferentes perspectivas (clientes, usuarios, desarrolladores y expertos en pruebas) que cooperan para escribir unas pruebas de aceptación antes de implementar la funcionalidad asociada (Agile Alliance, 2015b).

Tiene en común con TDD el hecho de que las pruebas se definen antes de escribir el código y que estas reflejan ciertas expectativas sobre su funcionamiento.

La diferencia está en que estas pruebas se definen en estrecha colaboración con el cliente, como se muestra en la Figura 7, de manera que ayudan a comprender las historias de usuario y a capturar su esencia en pruebas ejecutables. Podemos considerar las pruebas de aceptación como «requisitos ejecutables».

A diagram of distilling process

Description automatically generated

Figura 7. Modelo de ciclo de vida ATDD. Fuente: Hendrickson, 2008.

Veamos las fases de este ciclo de vida ATDD (Hendrickson, 2008):

* *Discuss* o discutir los requisitos. En esta fase los desarrolladores interrogan al usuario o cliente, o a su representante (el *Product Owner*) sobre las características de las historias de usuario elegidas para esta iteración. Es un punto en el que pueden surgir problemas no anticipados sobre la complejidad de una funcionalidad aparentemente sencilla.
* *Distill* o materializar las pruebas utilizando algún framework de desarrollo. En este caso emplearemos la herramienta más adecuada para automatizar las pruebas en función de la tecnología y el tipo de proyecto que estemos desarrollando. Algunos frameworks populares son FIT, FITNesse, Concordian o Robot Framework.
* *Develop* o desarrollar el código hasta conseguir que las pruebas definidas se ejecuten correctamente. Durante esta fase es posible aplicar la técnica TDD para implementar incrementalmente cada historia de usuario.
* Demo o demostración final. Al final de cada iteración se demuestra el incremento realizado, incluyendo la comprobación de que todas las pruebas de aceptación definidas inicialmente se ejecutan satisfactoriamente.

10.6. Behavior Driven Development (BDD)

El desarrollo dirigido por comportamiento —*Behavior Driven Development* (BDD)— es una síntesis o refinamiento de prácticas que proceden del desarrollo dirigido por pruebas (TDD) y del desarrollo dirigido por pruebas de aceptación (ATDD).

Esencialmente, ofrece un proceso guiado para escribir buenas pruebas automatizadas. BDD mejora a TDD y ATDD con las siguientes tácticas (Agile Alliance, 2015c):

* Está claramente orientado a los objetivos de negocio.
* Se piensa «desde fuera hacia dentro», priorizando los comportamientos que claramente aportan valor a los objetivos de la aplicación. De esta manera, se intenta minimizar el despilfarro de recursos, pues el desarrollo se centra en la elaboración de código que satisface unos requisitos funcionales concretos.
* Mejora el proceso de comunicación. Viene acompañado de herramientas que permiten describir los comportamientos en un lenguaje comprensible para todos los *stakeholders* (cliente y usuarios, expertos en el dominio de aplicación, ingenieros de pruebas y desarrolladores), tengan o no formación técnica. De esta manera, se mejora el proceso de comunicación.

Podemos considerarlo una extensión del TDD, menos centrada en la implementación y más orientada hacia el escenario (el comportamiento general de la funcionalidad desde el punto de vista del problema que resuelve).

Uno de los padres de esta técnica fue Dan North, creador del *framework* BDD JBehave (JBehave, 2021), quien durante una conferencia celebrada en Londres en 2009 describió está técnica del siguiente modo: «BDD es una metodología ágil de segunda generación, con un enfoque de fuera hacia dentro, que facilita la extracción de información de múltiples *stakeholders* y a diferentes escalas, y altamente automatizada» (North, 2009).

A pesar de sus beneficios, puede resultar una técnica más compleja de implementar que TDD, y es recomendable solo para desarrolladores familiarizados con esta última. Como veremos, esta técnica viene acompañada de herramientas específicas que permiten trasladar las historias de usuario a una notación más formal, pero aún próxima al lenguaje natural, de manera que los comportamientos esperados quedan claramente definidos.

En el artículo de Dan North, *Introducing BDD* (2006), se puede encontrar una interesante introducción a esta técnica, donde se hace énfasis en el hecho de que lo importante en BDD es el modelado del comportamiento del producto desde el punto de vista del usuario.

La notación «*Given-When-Then*»

Dentro de este enfoque se ha originado la notación «*Given-When-Then*» (GWT), que facilita la comprensión de los comportamientos y la implementación de las pruebas.

Se trata de una plantilla que facilita la escritura de las pruebas de aceptación para una historia de usuario y que es seguida por las herramientas que dan soporte a esta técnica.

Se compone de tres partes (Agile Alliance, 2015d):

* *Given* (dado que). Indica el contexto o punto de partida de la acción que se indicará a continuación.
* *When* (cuando). Indica la acción que realiza el usuario o, en general, el evento que se produce dentro o fuera de la aplicación y que desencadena una respuesta.
* *Then* (entonces). Se indica el resultado de la acción realizada en ese determinado contexto como un conjunto de consecuencias observables que se obtienen.

Veamos un ejemplo de prueba definida con la notación:

Ejemplo de prueba definida con la notación

* Dado que la cuenta de mi banco tiene crédito y no he realizado retiradas de efectivo recientemente.
* Cuando intento retirar una cantidad inferior al límite de mi tarjeta.
* Entonces la retirada debería completarse sin errores o avisos.

Algunos *frameworks* de pruebas que soportan este enfoque son JBejave, RSpec o Cucumber. El *framework*Cucumber, facilita un enfoque BDD con múltiples lenguajes de programación. Esta herramienta se creó inicialmente como soporte para el lenguaje de programación Ruby y, más específicamente, como complemento del *framework* RSpec (Rose, Wynne, y Hellesøy, 2015). Actualmente, existen diferentes implementaciones para otros muchos lenguajes, como Java, PHP o Python. Todas estas implementaciones comparten la utilización del lenguaje Gherkin para definir los diferentes casos de prueba en la forma de características, escenarios y pasos.

Cucumber se apoya fuertemente en el uso de expresiones regulares para mapear los pasos definidos en un escenario con sus correspondientes implementaciones funcionales en un lenguaje concreto, de manera que se automatiza la invocación de este código que, en última instancia, ejecuta las pruebas (Cucumber Docs, 2019).

10.7. La calidad del software

Según la Real Academia Española de la Lengua, la calidad es la «propiedad o conjunto de propiedades inherentes a algo, que permiten juzgar su valor» (RAE, 2021, definición 1).

Una definición más técnica, contenida en la norma ISO 9000:2005, dice que se trata del «grado en que un conjunto de características inherentes de un objeto cumple con sus requisitos» (ISO, 2015). El hecho de que las características sean «inherentes» implica que existen de facto en el objeto y, por tanto, pueden ser medidas, mientras que un requisito es una «necesidad o expectativa que se declara sobre ese objeto», generalmente con carácter obligatorio.

Cuando hablamos de la calidad del producto *software* o calidad del *software,* la norma ISO/IEC 25000:2014 indica que es la «capacidad del producto software para satisfacer las necesidades expresadas o implícitas, cuando se utiliza en condiciones específicas» (ISO, 2014a).

Vemos, por tanto, que la calidad depende de los atributos del objeto (que deben ser observables) y siempre está relacionada con las expectativas del usuario y condiciones de utilización concretas. Estas necesidades o expectativas son precisamente los requisitos del producto.

Es más fácil identificar los sistemas de baja calidad como aquellos que no cumplen la función que se espera de ellos, que no se utilizan porque resultan demasiado complejos o que no pueden mantenerse cuando el equipo que los desarrolló desaparece. Por el contrario, la identificación de la calidad es más compleja, pues precisa de la definición de métricas concretas que permitan evaluar en qué medida las expectativas sobre el producto son alcanzadas.

En la norma ISO/IEC 25010 se presentan tres enfoques del concepto de calidad que se relacionan en un ciclo de vida de calidad, de manera que cada nivel tiene influencia en los siguientes, como se observa en la Figura 8. Mientras que la calidad del propio proceso de desarrollo influye claramente en los atributos internos y externos del producto, estos tienen un impacto final en las características del producto una vez que es utilizado en diferentes contextos de operación.

A diagram of software quality attributes

Description automatically generated

Figura 8. Relación entre la calidad del proceso, la calidad interna, la calidad externa y la calidad en uso del producto software (mientras que los tres primeros elementos son controlables por el proveedor o fabricante del producto, la calidad en uso depende de la percepción que el usuario tiene durante su utilización). Fuente: ISO, 2011.

Del mismo modo, las metas que se fijen para los atributos de calidad del producto van a determinar las metas fijadas y las características deseadas para el proceso de desarrollo, que dará como resultado ese producto con ese nivel de calidad exigido.

* Calidad interna: es observable a partir de características intrínsecas del propio producto como el código fuente. Aquí encontramos atributos como: la estructura del código, la mantenibilidad, la complejidad, el acoplamiento, la escalabilidad o la reusabilidad. Todos estos aspectos solo pueden ser evaluados mediante inspecciones y pruebas estáticas del *software,* analizando la propia estructura del código fuente, sus componentes, y el diseño y arquitectura del sistema.
* Calidad externa: es observable a partir del comportamiento del producto y puede ser evaluada mediante pruebas de *software.* Encontramos aspectos como la corrección, la eficiencia y los costes de mantenimiento. Aquí es donde las pruebas de *software* tienen utilidad, pues permiten ejecutar el producto y comprobar si se verifican determinados requisitos de calidad externa.
* Calidad en uso: es observable durante la utilización del producto por parte del usuario, en un entorno de pruebas o de producción.

Cuando construimos *software* intentamos que la mejora del proceso repercuta en la calidad interna, que esta a su vez lo haga en la calidad externa y que, finalmente, la calidad en uso para cada uno de los contextos de uso sea buena.

Las dos primeras categorías son las que, de alguna manera, puede evaluar directamente el fabricante sin la intervención del usuario.

Sin embargo, aparece un dilema que viene representado por el «iceberg de la calidad del *software*» (Figura 9), que marca una clara diferencia entre las pruebas del *software* (*software testing*) y el aseguramiento de la calidad (*quallity assurance*), o entre lo «visible» y lo «invisible» (Bloom, 2011).

A blue iceberg with white text

Description automatically generated

Figura 9. El iceberg de la calidad del software. Fuente: McConnell, 1993.

Mientras que las pruebas tradicionales están dirigidas a verificar la calidad del *software* desde el punto de vista de su funcionamiento, muchas veces solo pueden llegar a determinar la presencia de errores o la ausencia de calidad, pero no pueden garantizar un *software* libre de errores o detectar posibles causas más profundas relacionadas con atributos de calidad interna.

Actualmente todos los aspectos relacionados con la calidad del proceso de desarrollo de *software* y la calidad del propio producto están reclamando cada vez más atención en el mundo de la ingeniería del *software.*

Son dos aspectos que están claramente interconectados: la mejora de los procesos de construcción de *software* afecta a la calidad del resultado, mientras que las mejoras introducidas en el propio producto pueden facilitar los procesos que soportan su desarrollo.

Las principales iniciativas se han centrado en la mejora de los procesos, pero no debemos olvidar que, finalmente, de lo que se trata es de mejorar la calidad del producto construido. Algunos autores han señalado que cualquier estándar o proceso de certificación debería estar centrado principalmente en el producto y no tanto en el proceso. «Las evaluaciones deberían basarse en evidencias directas sobre los atributos del producto, y no en evidencia circunstancial sobre el proceso» (Maibaum y Wassyng, 2008, p. 91).

Otros autores señalan que existe poca evidencia de que la adopción de procesos estandarizados garantice en sí misma la obtención de buenos productos. Los procesos estandarizados garantizan la uniformidad de los resultados obtenidos, pero se corre el peligro de institucionalizar prácticas perjudiciales que no estén en realidad contribuyendo al objetivo final.

Por todos estos motivos, se ha detectado la necesidad de contar con normas y marcos metodológicos que faciliten la medición de la calidad del producto *software,*su análisis y la elaboración de los resultados en informes. La nueva familia de estándares ISO/IEC 25000 se centra precisamente en este interés, también denominada SQuaRE (iso25000.com, 2021a): *System and Software Quality Requirements ad Evaluation*(requisitos y evaluación de la calidad del *software* y el sistema).

10.8. Modelos de calidad del producto software

En esta sección se van a revisar algunas de las principales normas y modelos que se ocupan de valorar el producto *software,* teniendo en cuenta un conjunto de atributos que lo caracterizan. La mayoría de los modelos de calidad del *software* se estructuran como se muestra en la Figura 10.

A diagram of a company

Description automatically generated

Figura 10. Estructura de la calidad del software. Fuente: Callejas, Alarcón y Álvarez, 2017.

* Los factores de calidad, o atributos externos, son las características que determinan la calidad desde el punto de vista del usuario. Aglutinan las expectativas que tiene el usuario sobre los beneficios que obtendrá del producto.
* Los criterios de calidad, o atributos internos, son características del propio producto asociadas a cada factor.
* Las métricas asociadas a cada criterio de calidad son medidas cuantitativas concretas que permiten valorar el grado de presencia de un atributo.

Los diferentes modelos de calidad del producto *software* se pueden clasificar según tengan un enfoque de modelo de calidad fijo, a medida o mixto (Calero, Piattini y Moraga, 2010):

* Los modelos fijos ofrecen un catálogo de factores de calidad para emplear como base de la evaluación. Generalmente facilita una jerarquía amplia de factores, criterios y métricas, de los que se pueden seleccionar los elementos más adecuados para un proceso de evaluación concreto. A esta categoría pertenecen los modelos de McCall (McCall *et al.*, 1977b, 1977a), Boehm (Boehm, 1978) y FURPS (Grady y Caswell, 1987) con un enfoque más industrial.
* Los modelos a medida se definen *ad hoc* para cada proyecto partiendo de los objetivos concretos que se desea alcanzar.
* Los modelos mixtos intentan combinar las ventajas de los modelos anteriores. Ofrecen un conjunto de factores de calidad de alto nivel reutilizables en todos los proyectos, pero refinados para cada caso particular. A esta categoría pertenece el estándar ISO/IEC 9126-1.

La Tabla 3 recoge las principales ventajas e inconvenientes de las dos primeras categorías, lo cual da una idea de los potenciales beneficios de un modelo mixto. En el trabajo de Callejas *et al.* (2017), se puede encontrar una revisión de algunos de los modelos de calidad más conocidos, junto con una recopilación de diferentes implementaciones a nivel mundial.

A blue and white box with black text

Description automatically generated

Tabla 3. Ventajas e inconvenientes de los diferentes tipos de modelos de calidad de software. Fuente: elaboración propia.

En este apartado se revisarán únicamente algunos de los estándares internacionales más directamente relacionados con SQuaRE.

ISO/IEC 9126:2001

Esta norma establece un modelo de calidad para el producto *software.* La primera versión apareció en 1991, aunque fue sustituida en 2001 por dos estándares relacionados, el ISO/IEC 9126 para la calidad del *software* y el ISO/IEC 14598 para la evaluación de productos de *software.*

Tanto esta norma como su sucesora, ISO 25000 (SQuaRE), contemplan la calidad del producto desde tres perspectivas:

* Una interna: relacionada con la calidad del código.
* Otra externa: relacionada con las características en ejecución.
* Otra en uso: más relacionada con la perspectiva del usuario.

Se trata de un modelo mixto por definir «una jerarquía de factores de calidad clasificados en características, subcaracterísticas y atributos» (Calero *et al.*, 2010, p. 294).

Se ofrece un conjunto de factores de partida compuesto por seis características comunes a las vistas interna y externa (funcionalidad, fiabilidad, eficiencia, usabilidad, mantenibilidad y portabilidad), y veintisiete subcaracterísticas.

ISO/IEC 14598

Esta norma está directamente relacionada con ISO 9126 y se encarga de establecer los procesos para realizar la evaluación de calidad del *software.* Su primera versión salió a la luz en 1999. Las normas ISO 9126 e ISO 14598, a pesar de estar relacionadas entre sí, tienen múltiples inconsistencias, por este motivo se ha desarrollado el conjunto de normas ISO/IEC 25000, que aglutina en una misma familia tanto el modelo de calidad del producto *software* como las guías para realizar el proceso de evaluación.

CISQ

El *Consortium for IT Software Quality* (CISQ), es una organización fundada por el OMG (*Object Management Group*) y el SEI (*Software Engineering Institute*) para desarrollar estándares que permitan automatizar la medida del tamaño del *software* y la calidad estructural del código fuente (CISQ, 2018).

A diagram of a product software

Description automatically generated

Figura 11. Relación entre los estándares ISO/IEC 9126 e ISO/IEC 14598. Fuente: Calero, Piattini y Moraga, 2010.

A la hora de analizar la calidad del código podemos considerar que CISQ se basa en SQuaRE, pues elige cuatro de las ocho características propuestas en el modelo de calidad de *software* y propone maneras de automatizar su medida. Estas características son la fiabilidad, la eficiencia de desempeño, la seguridad y la mantenibilidad, por ser, en opinión del OMG y el SEI, las más importantes y susceptibles de poder medirse de manera automática con herramientas específicas.

10.9. Familia ISO/IEC 25000

La familia de estándares ISO/IEC 25000 se centra en el aspecto de evaluar la calidad del producto *software.* Toma el sobrenombre de SQuaRE (iso25000.com, 2021a): *System and Software Quality Requirements and Evaluation* (requisitos y evaluación de la calidad del software y del sistema).

ISO/IEC 25000 aparece para sustituir y evolucionar un conjunto de normas anteriores basadas en ISO/IEC 9126 (*Software Engineering-Product Quality*), que ofrecía un modelo de calidad del producto *software,* y en ISO/IEC 14598 (*Information Technology-Software Product Evaluation*), que se centra en la evaluación del producto *software.*

El objetivo de SQuaRE es la construcción de un marco común de referencia para evaluar la calidad del producto *software.*

A su vez, ISO/IEC 25000 se descompone en cinco grandes divisiones. Cada una de ellas está constituida por diferentes normas que tratan aspectos específicos, los cuales se resumen en la Tabla 4.

A table of quality control

Description automatically generated with medium confidence

Tabla 4. Diferentes componentes del conjunto de normas ISO/IEC 25000. Fuente: ISO, 2021.

Además de estas divisiones, se contempla la futura extensión de SQuaRE con nuevas normas (de la 25050 a la 25099 para dominios de aplicación específicos) y recoge así sus particularidades concretas. Por ejemplo:

* ISO/IEC 25051:*Requirements for quality of Ready to Use Software Product* (RUSP) *and instructions for testing.*Contiene requisitos de calidad y requisitos para realizar las pruebas de productos RUSP (ISO, 2014b).
* ISO/IEC 25062:2006: *Common Industry Format* (CIF) *for usability test reports.* Está dirigido a expertos en usabilidad para guiarles en la generación de informes de pruebas de usabilidad (ISO, 2006).

Es importante recordar que SQuaRE describe qué se debe evaluar, pero no especifica cómo hacerlo. No detalla los umbrales que deben emplearse junto a las métricas ni la manera de agruparlas para asignar una única puntuación de calidad a un producto (Rodríguez y Piattini, 2014).

De todas las normas anteriores, podemos destacar la ISO/IEC 25010, por definir el modelo general de calidad del producto y el conjunto de atributos que deberían evaluarse; la ISO/IEC 25012, por hacer lo mismo para el caso particular de los datos; y la ISO/IEC 25040, puesto que proporciona un modelo de referencia general para el proceso de evaluación.

ISO/IEC 25010: modelo y características de calidad

La norma ISO/IEC 25010:2011 define los modelos fundamentales que proporcionan un conjunto de características susceptibles de ser evaluadas para determinar la calidad del producto *software.*

Por un lado, se define un modelo de calidad en uso compuesto por cinco características (efectividad, eficiencia, satisfacción, ausencia de riesgos y cobertura de contexto) y, por otro, un modelo de calidad del producto definido por otras ocho características (Figura 12). Ambos modelos se complementan con el modelo de calidad de los datos que proporciona la norma ISO/IEC 2012:2008.

A blue and white table with white text

Description automatically generated

Figura 12. Atributos de calidad según el modelo de calidad de producto de la norma ISO/IEC 25010:2011. Fuente: ISO, 2011.

ISO/IEC 25012: la calidad de los datos

La calidad de los datos viene determinada por el grado en que se satisfacen sus propios requisitos. El modelo de calidad de datos está compuesto por un total de quince características que se clasifican en dos grandes categorías:

* Calidad de datos inherente: tiene que ver con el grado en que las características de los datos tienen potencial intrínseco para satisfacer las necesidades del sistema.
* Calidad de datos dependiente del sistema: grado en que la calidad de los datos es alcanzada y preservada durante su utilización en un sistema informático. Depende del dominio tecnológico de uso y de las capacidades de los componentes del sistema.

ISO/IEC 25020: medición de la calidad del *software*

La norma ISO/IEC 25020:2016 presenta un modelo de referencia para la medida de la calidad del producto *software.*

Describe la relación entre el modelo de calidad, sus características, los atributos del producto y sus correspondientes medidas de calidad, funciones y métodos de medición (ISO, 2007).

La norma también proporciona ejemplos de criterios para seleccionar las medidas de calidad. La Figura 13 representa las relaciones entre las diferentes normas que componen la división de medición de la calidad.

A diagram of quality measurement

Description automatically generated

Figura 13. Estructura de la división de «Medición de la calidad» (2502x) de SQuaRE. Fuente: ISO, 2007.

ISO/IEC 25040: evaluación del *software*

La norma ISO/IEC 25040:2011 contiene requisitos y recomendaciones para la evaluación de la calidad del producto *software.* Puesto que está dirigida a todo aquel responsable de la evaluación del producto, resulta apropiada para desarrolladores, adquisidores y también evaluadores independientes de productos.

Define un modelo de proceso de evaluación formado por cinco actividades principales, como se muestra en la Figura 14.

A diagram of a diagram

Description automatically generated

Figura 14. Actividades y tareas definidas para el proceso de evaluación según ISO/IEC 25040:2011. Fuente: elaboración propia.

El proceso de certificación mediante SQuaRE

La certificación de calidad de un producto mediante SQuaRE permite, por un lado, a las empresas que fabrican *software,* conocer la calidad de sus productos y, por otro, a las empresas compradoras, les ayuda a decidirse por la solución más adecuada.

Los principales beneficios que ofrece la obtención de una certificación son los siguientes (AENOR, 2018; Fernández, Rodríguez y Piattini, 2013):

* Permite a los fabricantes de *software* diferenciarse de sus competidores.
* Permite establecer acuerdos de servicio con un proveedor, definiendo parámetros de calidad que deben cumplirse al entregar el producto.
* Facilita la detección de defectos en un producto y su mejora antes de la entrega, ahorrando costes en la fase de mantenimiento posterior.
* Facilita la evaluación y control del rendimiento del producto.

El proceso de evaluación y certificación conforme a la norma ISO/IEC25000 consta de seis pasos, como se aprecia en la Figura 15:

A diagram of quality evaluation

Description automatically generated

Figura 15. Proceso de evaluación y certificación del producto *software.* Fuente: iso25000.com, 2021b.

1. La organización interesada solicita una evaluación a un laboratorio acreditado. Deben indicar qué características del producto desean evaluar. El laboratorio envía un contrato de evaluación especificando las condiciones del servicio. Si es aceptado, la organización envía el *software* al laboratorio, que realiza las pruebas empleando un *framework* (modelo, proceso y herramientas) basado en SQuaRE y acreditado por ENAC (Entidad Nacional de Acreditación).
2. El resultado de la evaluación se envía a la organización solicitante. Si es desfavorable, la organización puede recurrir a la ayuda de un consultor de calidad que le ayude a mejorar su producto hasta que consiga superar la evaluación. El proceso finaliza cuando se obtiene una evaluación favorable por parte del laboratorio acreditado.
3. La organización solicita el certificado a la entidad de certificación (en este caso AENOR), indicando la referencia de la evaluación realizada por el laboratorio acreditado.
4. La entidad de certificación solicita el resultado de la evaluación al laboratorio.
5. El laboratorio proporciona la información a la entidad de certificación.
6. Finalmente, la organización certificadora visita al solicitante para comprobar el producto y sus características, y emite un informe junto con el certificado que reconoce la calidad del producto evaluado.

10.10. La mejora del proceso de software

La mejora del proceso de *software* abarca un conjunto de actividades que buscan optimizar los procesos de construcción del *software* con el objetivo final de obtener un producto de mayor calidad, más rápido y con menor coste (Pressman, 2010).

Estas actividades pueden adoptar dos enfoques posibles (Sommerville, 2011):

* La madurez de procesos. Busca introducir en la organización un conjunto de buenas prácticas conocidas por su beneficio en la mejora de los procesos. En este caso, el nivel de capacidad de un proceso indica en qué medida incluye una serie de buenas prácticas recomendadas.
* El enfoque ágil. Las técnicas ágiles adoptan enfoques iterativos del proceso y se centran en la entrega temprana y rápida de funcionalidades, con el objetivo de obtener retroalimentación del cliente, que repercutirá, finalmente, en la mejor calidad del producto desarrollado.

Aparentemente, ambos enfoques resultan contrapuestos. Mientras que el primero se ocupa de facilitar el desarrollo de planes precisos que contemplen una serie de prácticas necesarias para la buena calidad del proceso, el segundo da menos importancia a la planificación y a la estandarización de procesos, y busca la adaptación (agilidad) como mejor estrategia para resolver los problemas y conseguir los objetivos del proyecto.

En el primer caso, se pueden introducir algunas prácticas que suponen una sobrecarga de trabajo (no necesariamente relacionadas con la programación) y, en el segundo, se intenta eludir toda carga burocrática y el foco está en el desarrollo.

Sin embargo, no debemos olvidar que algunas técnicas ágiles, como XP, hacen énfasis precisamente en determinadas prácticas de ingeniería (buenas prácticas) al conocerse que redundan en un beneficio de la calidad final del producto. En este sentido, un equipo de desarrollo XP es «maduro» en la medida en que es capaz de adoptar el conjunto de prácticas recomendadas.

En términos generales, las prácticas ágiles resultan beneficiosas en proyectos pequeños y medianos, en los que es fundamental focalizar los recursos en la entrega del producto y en la satisfacción del cliente, y los procesos son adaptables en función de las necesidades concretas.

En proyectos grandes, construcción de sistemas críticos y cuando el equipo integra desarrolladores de varias compañías distribuidos geográficamente y con culturas diferentes, el enfoque basado en la mejora y estandarización de procesos puede resultar más adecuado.

Factores que afectan a la calidad del producto *software*

Los primeros esfuerzos en el campo de la ingeniería orientados a la mejora de los procesos tuvieron su origen en el ámbito de la fabricación. Se desarrollaron técnicas como el control estadístico de calidad(centrado en la medida del número de defectos en los productos y su relación con el proceso de fabricación).

Este enfoque no resulta del todo apropiado, en el caso del desarrollo de *software,* por dos motivos:

* En los procesos de fabricación hay una clara relación entre el proceso y el producto. Son procesos estandarizados en los que se fabrican grandes volúmenes de elementos iguales. La correcta configuración de una máquina y la definición de secuencias de procesamiento necesarias resultan fundamentales, quedando a menudo fijadas en el diseño de planta de la fábrica. En el caso del desarrollo *software,* esta restricción no existe y, además, cada producto es único.
* En el ámbito de la fabricación es fácil establecer métricas de calidad para un producto (número de defectos por unidad de superficie, resistencia a la tracción, etc.). El *software* es un producto intangible y, a menudo, su calidad final viene determinada por un conjunto de atributos más subjetivos.

En términos generales, la calidad de un producto *software* está influenciada por los elementos representados en la Figura 16: calidad del proceso, tecnología empleada, calidad del personal (cualificación) y recursos disponibles (tiempo, presupuesto, etc.). La influencia relativa de cada uno viene, a su vez, determinada por otros factores como el tamaño del proyecto y sus características.

A diagram of a product

Description automatically generated

Figura 16. Factores que influyen en la calidad de un producto *software*. Fuente: Sommerville, 2011.

* En proyectos grandes, con muchos participantes, tiende a ser más determinante la calidad del proceso empleado y la correcta calendarización y estimación del coste del proyecto. Los procesos definidos permiten asignar responsabilidades, controlar el proceso, y establecer los cauces de comunicación adecuados y claros. Los errores de planificación pueden resultar desastrosos y difíciles de corregir.
* En proyectos pequeños, con equipos de desarrollo reducidos, donde a menudo los recursos son más limitados, resulta fundamental la experiencia del equipo y el correcto aprovechamiento de las herramientas tecnológicas. La falta de procesos estandarizados y claros se suple con la experiencia y los conocimientos del equipo, mientras que, al no poder dedicarse recursos a actividades de gestión, es necesario contar con herramientas tecnológicas que maximicen el resultado del tiempo empleado en desarrollo.

Recordemos que un proceso es un conjunto de actividades ejecutadas de una determinada manera para obtener un propósito concreto. En el esquema de la Figura 8 es un elemento más, pero también es el «pegamento» que une al resto.

La mejora de procesos asume la idea de que la calidad de un producto o sistema depende fundamentalmente de la calidad de los procesos empleados.

También es cierto que solo si el proceso seguido es el adecuado en cada caso, este repercutirá positivamente en la calidad del producto final. Muchas veces se institucionalizan prácticas y procesos en una organización que, al no ser los más apropiados, ocasionan que la mala calidad también se institucionalice en el producto *software.*

10.11. Modelos de madurez - CMMI

Los modelos de madurez se utilizan en el contexto de procesos de mejora del *software,* proporcionando indicadores sobre la «madurez de los procesos» o, dicho de otra manera, sobre su calidad o «nivel de capacidad». Esta valoración se realiza habitualmente empleando escalas ordinales, de manera que es posible asignar puntuaciones numéricas en función del nivel de madurez considerado para cada proceso o sobre la organización en su conjunto.

Como todo modelo, se trata de representaciones simplificadas del mundo que, en este caso, contienen los elementos esenciales de los procesos eficaces. Estos modelos de madurez van acompañados de sus correspondientes marcos conceptuales para el proceso de mejora.

«Contienen los elementos esenciales de los procesos eficaces de una o más disciplinas y describen un camino evolutivo de mejora desde procesos ad hoc e inmaduros a procesos disciplinados y maduros con calidad y eficacia mejoradas» (SEI, 2010, p. 5).

Aunque no existe un enfoque único y el proceso debe adaptarse a la organización, un marco conceptual de mejora define, según Pressman (2010):

1. Un conjunto de características deseables para cada proceso. Estas características se basan en la observación de buenas prácticas a lo largo del tiempo.
2. Un método para evaluar el proceso. El marco conceptual define la manera de analizar los procesos y decidir en qué medida se ajusta a las características deseadas.
3. Un sistema de informes. Se proporciona una manera de resumir los resultados de las valoraciones.
4. Estrategia de apoyo. El marco también ofrece a la compañía procesos para acompañarla en la transición hacia la implementación del cambio.

Los marcos conceptuales más extendidos que permiten la mejora del proceso de *software* son el CMM y el CMMI del SEI (*Software Engineering Institute*). Sin embargo, existen algunas otras alternativas que vale la pena señalar (Pressman, 2010) y que aparecen representadas en una línea de tiempo en la Figura 17:

A line of numbers and letters

Description automatically generated

Figura 17. Cronología de diferentes modelos de calidad a nivel de proceso. Fuente: Callejas, Alarcón y Álvarez, 2017.

* ITIL (*Information Technology Infrastructure Library*), biblioteca de infraestructura de tecnologías de información. Proporciona un conjunto de buenas prácticas para la gestión de servicios de tecnologías de la información (TI), el desarrollo de tecnologías y las operaciones en general.
* SPICE (*Software Process Improvement and Capability Determination*) o determinación de mejora y capacidad del proceso de *software*es un marco conceptual de mejora, compatible con ISO 15504:2003 e ISO 12207. Es más flexible que el modelo del SEI e incluye niveles de madurez equivalentes.
* Bootstrap comenzó como un proyecto para adaptar el modelo del SEI a un amplio rango de compañías. Usa los mismos niveles de madurez que el CMMI, pero propone un modelo de proceso básico basado en el de la Agencia Espacial Europea (ESA) como punto de partida para la definición de procesos.

El modelo CMMI

El modelo de madurez de capacidades de *software* (CMM-SW) fue desarrollado por el Instituto de Ingeniería del Software de la Universidad Carnegie Mellon (*Software Engineering Institute,* SEI) durante la década de 1990, proporcionando un marco conceptual para abordar procesos de mejora del *software.*

A este le siguieron modelos para disciplinas específicas como el modelo de madurez de capacidades de personal (CMM-P) o el modelo de madurez de capacidades de ingeniería de sistemas (CMM-SE).

El marco CMMI (modelo de madurez de capacidad integrado) es una evolución del CMM que combina las disciplinas de *software* (CMM-SW) y sistemas (CMM-SE), «integrándolas» en un único marco conceptual.

En la Figura 18 se muestra, de manera esquemática, la evolución de los diferentes modelos de madurez y capacidad del SEI. Cuando apareció la versión 1.2 del CMMI se estaban desarrollando otros dos modelos integrados y, por este motivo, pasó a denominarse CMMI for Development (CMMI-DEV).

A diagram of software engineering

Description automatically generated

Figura 18. Evolución histórica de los modelos CMMI del SEI. Fuente: elaboración propia.

El modelo CMMI-DEV es muy complejo, pero según Sommerville (2011) podemos simplificarlo en los siguientes componentes:

* Un conjunto de veintidós áreas de proceso relacionadas con las actividades del proceso *software* y agrupadas en cuatro categorías (Tabla 5).
* Un conjunto de metas específicas asociadas a cada área de proceso. Cada meta define las características de sus actividades para que el proceso resulte efectivo.
* Un conjunto de metas genéricas aplicables a múltiples áreas de proceso. Describen características necesarias para institucionalizar los procesos de esa área.
* Un conjunto de prácticas específicas o actividades concretas que permiten lograr una meta determinada. Se trata de recomendaciones, y cada organización puede elegir la mejor manera para alcanzar las metas.
* Un conjunto de prácticas genéricas que describen actividades importantes para poder lograr metas genéricas y contribuir a la institucionalización de los procesos.
* Subprácticas. Son descripciones detalladas que ayudan a interpretar las prácticas específicas o genéricas.

A table with text on it

Description automatically generated

Tabla 5. Áreas de proceso identificadas en el CMMI (se han resaltado en negrita las áreas base o core). Fuente: Linders, 2016.

Como vemos en la Figura 19, el CMMI establece metas generales y específicas para cada una de las áreas de proceso, y propone una serie de prácticas para poder alcanzarlas. Cabe resaltar que CMMI no impone un modelo de proceso específico. Por lo tanto, es válido tanto en el contexto de las técnicas tradicionalescomo en el de las ágiles, y las prácticas propuestas pueden ser adoptadas en ambos casos (CMMI Institute, 2016; O’Neill, 2016).

A diagram of a process

Description automatically generated

Figura 19. Principales componentes del modelo CMMI-DEV. Fuente: elaboración propia.

Niveles de capacidad y niveles de madurez

Los niveles se utilizan en CMMI-DEV para «describir un camino evolutivo recomendado para una organización que quiera mejorar los procesos que utiliza» (SEI, 2010, p. 21). Vemos, por tanto, que es un concepto dinámico, no estático, orientado a facilitar la mejora de las organizaciones.

Existen dos tipos de niveles:

* Niveles de capacidad. Se refieren a la consecución de mejoras dentro de áreas de proceso individuales. Definen una escala de capacidad de cero a tres, como se muestra en la Tabla 6.

Como vemos, la valoración CMMI clasifica las áreas de proceso en una escala de cuatro niveles de capacidad (a partir de la versión 1.3 del CMMI desaparecen los niveles cuatro y cinco, y en mucha documentación se ve que se refieren a cinco niveles). Cada nivel indica la cantidad de metas alcanzadas por esa área en concreto. Alcanzar un nivel de capacidad en un área de proceso implica lograr todas las metas genéricas hasta ese nivel.

A blue and white list with black text

Description automatically generated with medium confidence

Tabla 6. Niveles de capacidad de áreas de proceso según el CMMI. Fuente: elaboración propia.

* Niveles de madurez. Se refieren a la consecución de mejoras en múltiples áreas de proceso de manera simultánea. En este caso, se define una escala de madurez de 1 a 5 (inicial, gestionado, definido, gestionado cuantitativamente y en optimización), como se muestra en la Tabla 7.

Finalmente podemos establecer una comparación entre niveles de capacidad y de madurez, como se muestra en la Tabla 8.

No tiene sentido definir un nivel de madurez 0 (incompleto) en una organización que tiene procesos incompletos. El término «inicial» hace referencia a que se trata de un punto de partida para la mejora de la organización, donde todos los procesos están, al menos, realizados.

La denominación de los niveles 2 y 3 coincide, y solo desde el punto de vista de la madurez de la organización se definen los niveles superiores 4 y 5.

A table of text with blue lines

Description automatically generated

Tabla 7. Niveles de madurez de la organización según el CMMI. Fuente: elaboración propia.

A blue and white table with white text

Description automatically generated

Tabla 8. Comparación entre niveles de capacidad y niveles de madurez. Fuente: SEI, 2010.

Los niveles de capacidad y los niveles de madurez dan lugar, respectivamente, a la representación continua y la representación por etapas.

* La representación continua permite una clasificación más fina de los procesos, calificándolos por áreas de proceso. Permite un proceso de mejora mejor adaptado a las necesidades específicas de la organización.
* La representación por etapas, compatible con el CMM, asigna un nivel de madurez de 1 a 5 al conjunto de los procesos que implementa una empresa.

La representación continua no arroja un único indicador de madurez global, sino un conjunto de valores o un perfil de capacidades que muestra la capacidad de varias áreas de proceso. Este perfil puede reflejar el estado actual de la empresa o la ruta de mejora elegida, como se muestra en la Figura 20.

A graph of a product

Description automatically generated with medium confidence

Figura 20. Ejemplo de perfil de nivel de capacidades según la representación continua para una empresa. El área sombreada muestra el perfil alcanzado, mientras que las barras no sombreadas muestran el perfil objetivo de la organización. Fuente: SEI, 2010.

Por su parte, la representación por etapas utiliza las mismas áreas de proceso, metas y prácticas que el modelo continuo, pero emplea los cinco niveles de madurez en vez de los cuatro niveles de capacidad. Se trata, por tanto, de una calificación global a nivel de un conjunto de procesos. Así, para alcanzar un determinado nivel de madurez, deben lograrse las metas genéricas asociadas a un conjunto de áreas de proceso, como se resume en la Tabla 9.

A table with text on it

Description automatically generated

Tabla 9. Niveles de madurez según las áreas de proceso requeridas. Fuente: Pressman, 2010; Linders, 2016.

CMMI y ágil

Como ya se ha mencionado, CMMI y las técnicas ágiles parecen estar en aparente contraposición.

* Las técnicas ágiles valoran más a los individuos y sus interacciones, el software funcional o la respuesta al cambio. Las técnicas ágiles se centran más en «cómo» hacer las cosas buscando la agilidad. También se ha comentado que algunas técnicas ágiles, como XP, valoran especialmente las prácticas utilizadas en el proceso de desarrollo.
* CMMI se ocupa de la mejora de procesos estandarizados, y ayuda a las empresas a poder planificarse mejor y a utilizar las herramientas adecuadas. CMMI se centra más en «qué» debe hacer una organización para definir, comprender y promover comportamientos que mejoren su rendimiento. CMMI es, ante todo, un conjunto de buenas prácticas, recopiladas y estructuradas en función de la experiencia acumulada a lo largo de muchos proyectos de desarrollo.

Parece, por tanto, que existe la posibilidad de reconciliar ambos enfoques (O’Neill, 2016). Pero recordemos antes que nada que ágil es un conjunto de técnicas, mientras que CMMI es un modelo de madurez. No son exactamente lo mismo, pero tienen cosas en común.

La clave para el éxito de las organizaciones es la adecuada combinación de «agilidad» y «estabilidad». Se trata de construir capacidades dinámicas sobre una estructura de procesos estable y una estructura organizacional sólida. Es posible identificar problemas concretos en los proyectos ágiles que están relacionados con áreas de proceso contempladas en CMMI (Tabla 10).

A table with text and images

Description automatically generated

Tabla 10. Ejemplos de relaciones entre posibles problemas en proyectos ágiles, las técnicas relacionadas y las áreas de proceso de CMMI que pueden dar soporte. Fuente: CMMI Institute, 2016.