Tema 8

Pruebas del software: TDD, ATDD y BDD

Ingeniería del Software Avanzada

Índice

[Esquema 3](#_Toc535213599)

[Ideas clave 4](#_Toc535213600)

[8.1. Introducción y objetivos 4](#_Toc535213601)

[8.2. Verificación y validación del software 5](#_Toc535213602)

[8.3. Las pruebas de software 8](#_Toc535213603)

[8.4. *Test Driven Development* (TDD) 16](#_Toc535213604)

[8.5. *Acceptance Test Driven Development* (ATDD) 29](#_Toc535213605)

[8.6. *Behavior Driven Development* (BDD) 31](#_Toc535213606)

[8.7. BDD y Cucumber 34](#_Toc535213607)

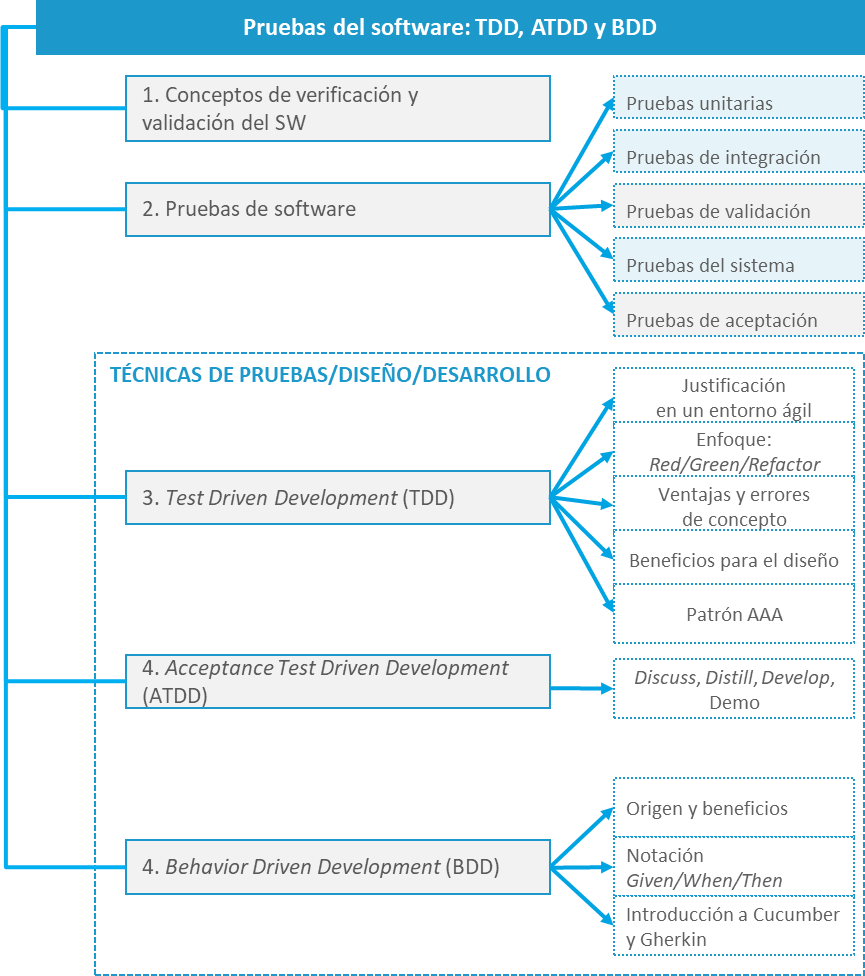
[8.8. Referencias bibliográficas 39](#_Toc535213608)

[A fondo 43](#_Toc535213609)

[Actividades 47](#_Toc535213610)

[Test 51](#_Toc535213611)

Esquema



Ideas clave

8.1. Introducción y objetivos

En este tema estudiaremos una de las técnicas más comúnmente empleadas y populares en la actualidad para guiar el proceso de desarrollo de software: ***Test Driven Development* (TDD),** **o Desarrollo dirigido por pruebas,** y algunas de sus extensiones, como **ATDD** (*Acceptance Test Driven Development*) o **BDD** (*Behavior Driven Development*).

Mediante estas técnicas, todo el proceso de construcción del software comienza precisamente por la definición de un conjunto de pruebas que tienen en cuenta la especificación de requisitos y que **alcanzan pleno sentido en el ámbito de las metodologías ágiles**, donde cada iteración comienza precisamente por la definición de estas pruebas.

Por todo ello, al comienzo del tema realizaremos un amplio repaso de los conceptos de verificación y validación del software, y el tipo de pruebas que abarcan.

En este tema pretendemos 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**, las fases que comprende y su relación con los tipos de pruebas estudiados anteriormente.
* **Conocer la técnica de desarrollo TDD** y los beneficios que supone para la mejora de la calidad del software desarrollado.
* Conocer los fundamentos de utilización del *framework* de pruebas **Cucumber** y el **lenguaje Gherkin** como implementación de la notación *Given/When/Then* típica de la técnica de desarrollo basada en comportamientos.

8.2. 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.

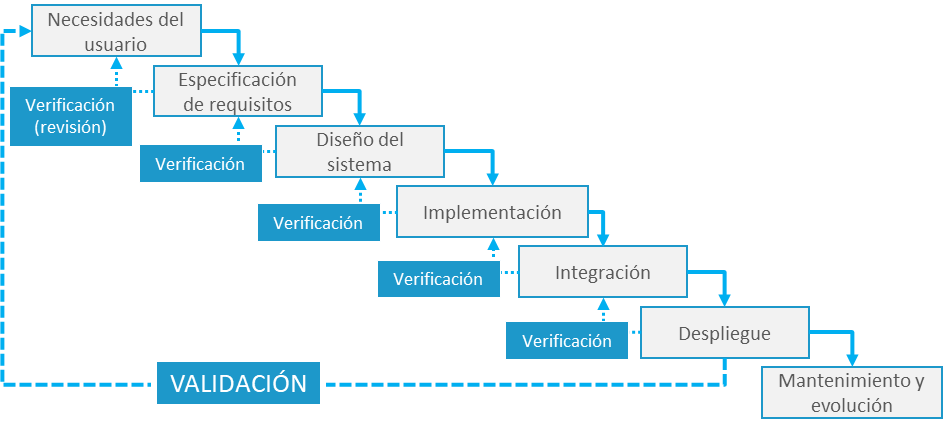


Figura 1. Diferencia entre los conceptos de verificación y validación del software

Estos conceptos aparecen reflejados en la figura 1.

* 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.

|  |  |  |
| --- | --- | --- |
|  | **VERIFICACIÓN** | **VALIDACIÓN** |
| **DEFINICIÓN** | Proceso en el que se evalúa un producto de trabajo en una fase de desarrollo para determinar que cumple con los requisitos definidos para esa fase. | Proceso de evaluación del software durante su construcción, o al final del proceso de desarrollo, para determinar que satisface los requisitos de negocio. |
| **OBJETIVO** | Asegurar que se construye el producto de acuerdo con los requisitos y especificaciones de diseño. | Asegurar que el producto cumple con las necesidades del usuario y que las especificaciones iniciales son correctas. |
| **PREGUNTA** | ¿Estamos construyendo correctamente el sistema? | ¿Estamos construyendo el sistema correcto? |
| **ELEMENTOS DE EVALUACIÓN** | Planificación, especificación de requisitos, diseños, código, casos de prueba. | El producto o sistema final. |
| **ACTIVIDADES** | * Revisiones * *Walkthroughs* * Inspecciones | * Pruebas de caja negra (funcionales) * Pruebas de caja gris * Pruebas de caja blanca (estructura) |
| **PROCESO** | Estático, no se ejecuta el código. | Dinámico, se prueba el producto. |
|  |  |  |

Tabla 1. Comparativa entre los procesos de verificación y validación del software.  
Fuente: Elaboración propia a partir de Sharma (2017) y STF (2011)

A pesar de esta definición de actividades orientativas que se recogen en la tabla anterior, 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.

8.3. Las pruebas de software

Podemos definir las **pruebas de software**, según 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, capítulo 1; Pressman, 2010, capítulo 17):

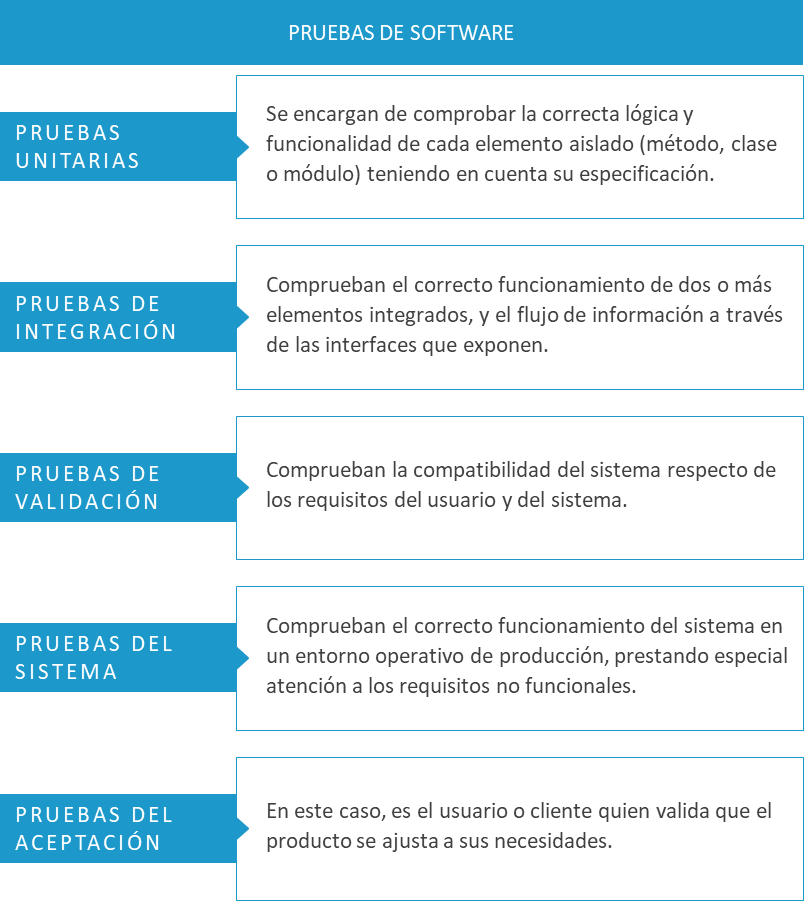


Figura 2. Estrategias de las pruebas de software

En un enfoque tradicional **estas pruebas se corresponden con momentos diferentes del ciclo de vida del desarrollo,** como se muestra en la figura 3.

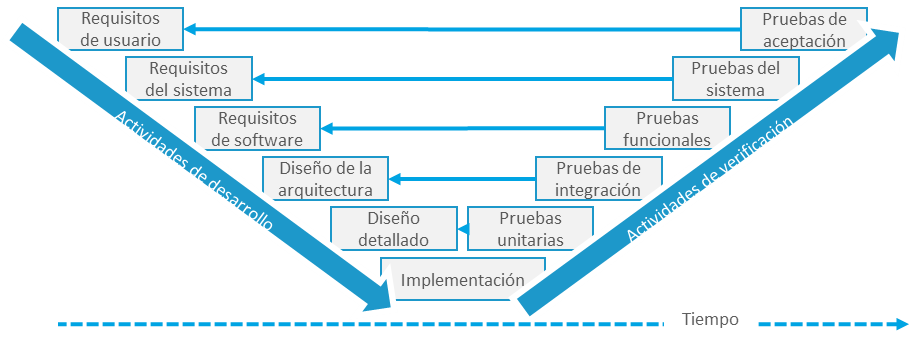


Figura 3. Modelo de proceso de software en V

El modelo de la figura 3, Modelo de proceso de software en V, puede ser considerado como una extensión del modelo en cascada, 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, que estudiaremos con algo de profundidad a continuación, 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.

|  |  |
| --- | --- |
|  | **PRUEBAS DE SOFTWARE** |
| **Equipo de desarrollo** | * Pruebas unitarias * De integración * Del sistema |
| **Equipo de desarrollo + Cliente** | * Validación * Aceptación |

Tabla 2. Perfiles que desarrollan las pruebas de software

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).

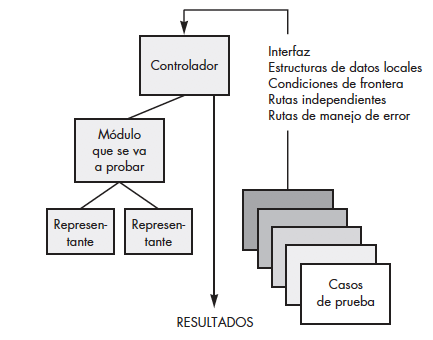
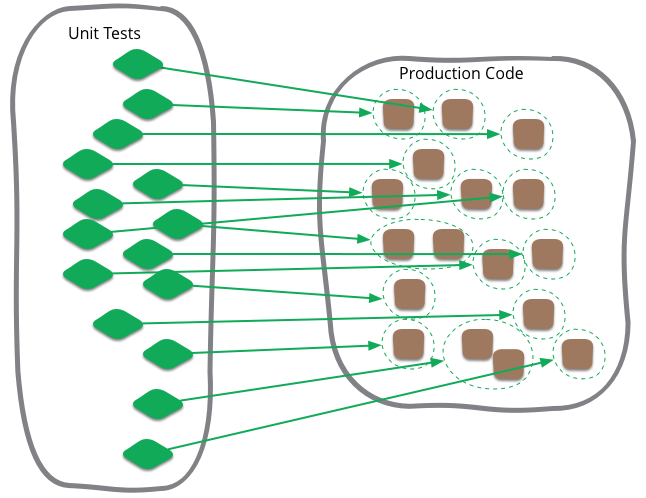


Figura 4. Las pruebas unitarias ejecutándose sobre un código en producción.  
Fuente: Fowler (2014) y Pressman (2010, p. 391).

Como vemos en la figura 4, **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 4 los representantes —*stubs*, en inglés— sustituyen otros módulos de los cuales depende el componente probado, simulando su comportamiento en el contexto de las pruebas unitarias.

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 (lo cual es conocido como «integración *big bang*») (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, p. 399).

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

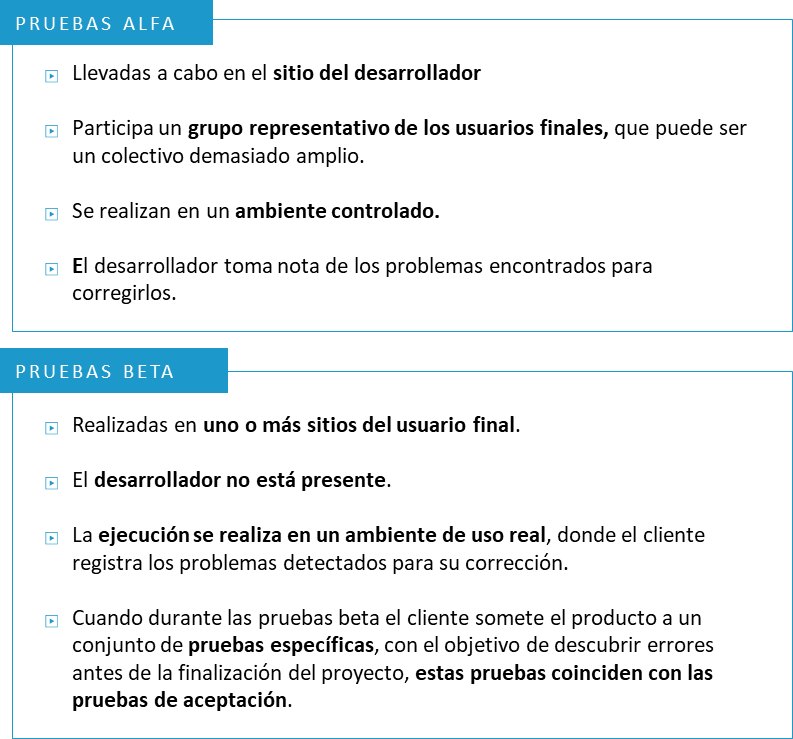


Figura 5. Tipos de pruebas de validación

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 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.**

Adquieren especial importancia en este contexto los **requisitos no funcionales** que, como sabemos, generalmente afectan al sistema en su conjunto e imponen restricciones que afectan 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, p. 24).

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 que 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**.

|  |  |  |
| --- | --- | --- |
| **FACTOR** | **PRUEBAS UNITARIAS** | **PRUEBAS DE ACEPTACIÓN** |
| **AUTOR** | Escritas por desarrolladores. | Escritas por el cliente. |
| **MOTIVACIÓN** | Su motivación es encontrar errores en el código. | Su motivación es demostrar las funcionalidades implementadas. |
| **RESULTADO** | Su ejecución satisfactoria no garantiza la ausencia de errores. | Su ejecución satisfactoria indica que el trabajo está finalizado. |
| **UTILIDAD** | Permiten identificar errores en unidades de código individuales.  Facilitan la documentación del código a bajo nivel. | Permiten verificar que la implementación es completa y correcta.  Facilitan un indicador del progreso de desarrollo durante el proyecto. |
| **CUANDO SE EJECUTAN** | Se escriben y ejecutan durante el desarrollo. | Se ejecutan antes del desarrollo y se ejecutan después. |
| **INFORMACIÓN DE PARTIDA** | Parten de una nueva capacidad que debe implementarse: un nuevo módulo, clase o método. | Parten de las necesidades del usuario: historias de usuario, casos de uso, requisitos, etc. |

Tabla 2. Comparativa entre las pruebas unitarias y las pruebas de aceptación.  
Fuente: Elaboración propia a partir de Gal–Oz y Balaban (2016)

8.4. *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 6.

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.

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.

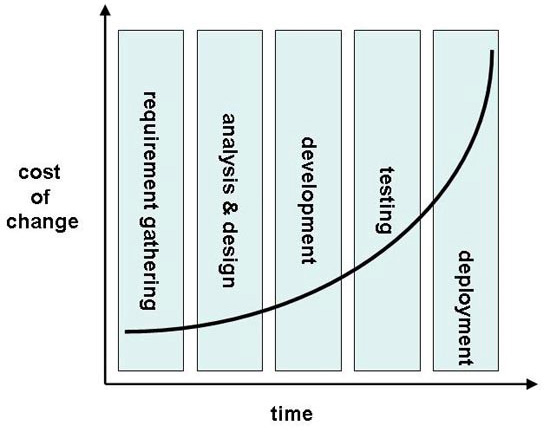


Figura 6. La tradicional curva del «coste del cambio» con el modelo de proceso en cascada en el fondo.  
Fuente: Murphy (2005), aunque el estudio original y la creación de esta famosa curva se debe a Barry Boehm (Boehm, 1981)

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 quienes 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 metodología ***eXtreme Programming***, de Kent Beck, y esencialmente consiste en **repetir de manera iterativa la siguiente secuencia de tres pasos**.

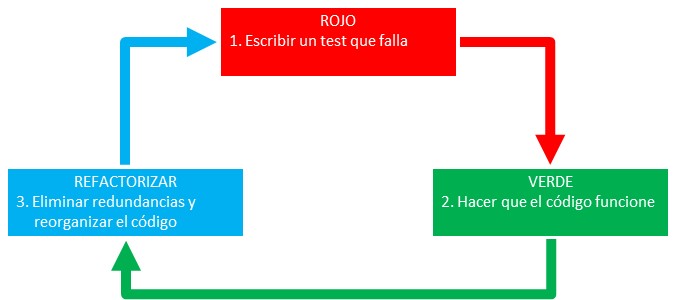


Figura 7. Ciclo de trabajo con TDD

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.

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 8 se muestra una comparación entre enfoques de desarrollo más tradicionales y el propuesto en TDD.

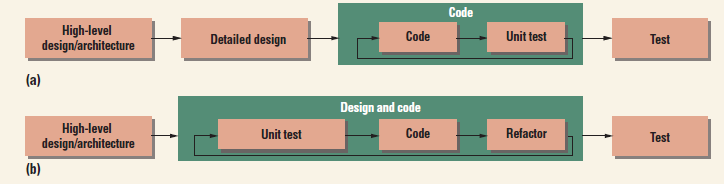


Figura 8. Flujo de desarrollo: (a) enfoque tradicional, con las pruebas al final y (b) *Test Driven Development*.  
Fuente: Janzen et al. (2008, p. 78)

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, Polytechnic, Obispo 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** (figura 9).

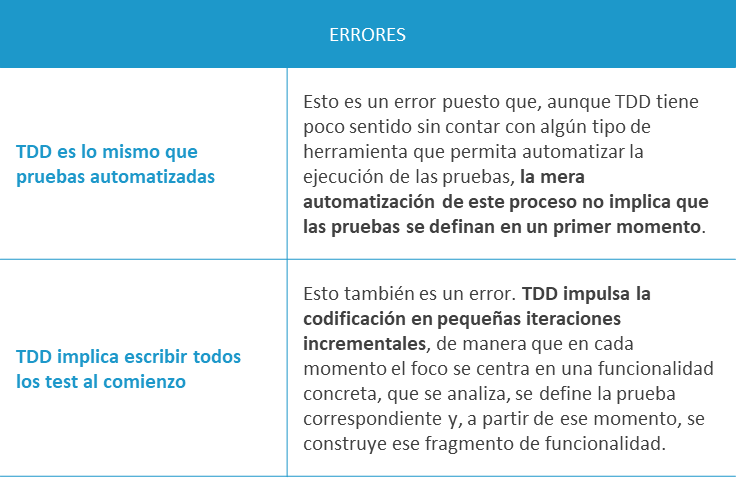


Figura 9. Errores de concepto de los desarrolladores

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, pues se pudo comprobar que:
  + Los equipos TDD desarrollaron **módulos y clases más ligeros**, con menos líneas de código.
  + Los equipos TDD también escribieron **métodos de tamaño más pequeño** en promedio.
  + Los equipos TDD **desarrollaron mayor número de clases**, pero con menor número de métodos en promedio.
* **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.

En este caso los autores emplearon las siguientes **métricas para evaluar el grado de acoplamiento interno del código**:

* + **Acoplamiento entre objetos** —*Coupling Between Objects* (CBO)—, que mide el número de conexiones entre objetos. En este caso no se obtuvieron diferencias significativas entre ambos enfoques.
  + ***Fan–out* por clase**, que mide el número de clases que utiliza otra. En este caso los resultados son similares a los del CBO, como cabría esperar.
  + **Número medio de parámetros por método**. En este caso la cifra es más elevada para el enfoque TDD, lo cual refleja que los programadores tienden a utilizar mayor número de parámetros para hacer que los métodos sean más fáciles de configurar y probar.
  + **Flujo de información** —*Information Flow* (IF)—, que se calcula con la siguiente fórmula:

**IF=〖"fan-in" 〗^2∙〖"fan-out" 〗^2.**

En este caso, el parámetro **fan-in** mide el número de clases que utilizan otra en concreto. Se concluyó en los equipos TDD este indicador era mayor. Teniendo en cuenta que los valores para el *fan–out* eran similares, ello podría indicar un mayor nivel de reutilización del código en este caso.

* **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**.

Es un parámetro difícil de medir, y la mayoría de las métricas investigan el grado en que los atributos de una clase se comparten entre los métodos. En este caso, eligieron, como métrica, la falta de cohesión de métodos (*Lack of COhesion of Methods*, LCOM5, Henderson-Sellers, 1995). Los resultaros mostraron que los equipos que no utilizaron TDD obtuvieron una mejor cohesión. Una posible explicación está en que la mayoría de las métricas de cohesión, incluyendo LCOM5, penalizan las clases que utilizan métodos de acceso a las variables internas, algo que es más habitual en desarrollos bajo TDD. Sin embargo, los autores no llegan a ninguna conclusión sólida al respecto.

* **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. En el estudio se utilizaron las siguientes **métricas de complejidad**:
  + **Complejidad ciclomática**, que indica el número de caminos independientes posibles en el código.
  + **El grado de anidamiento**, o profundidad de los bloques anidados dentro del código —*Nested Block Depth* (NBD)—.
  + **Métodos ponderados por clase** —*Weighted methods per class complexity* (WMC)—, que mide la suma de las complejidades ciclomáticas para todos los métodos de una clase.

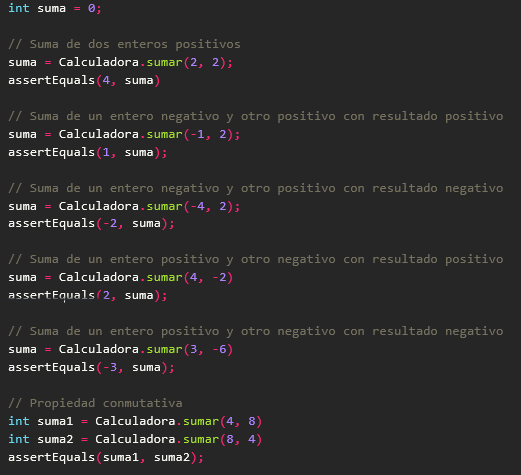
En el caso de las métricas de complejidad, se llegó a que, al ser analizadas las clases en su conjunto mediante la métrica WMC, los programadores TDD producían de manera consistente clases más simples en términos de número de ramas y número de métodos. En el caso de las otras dos métricas los resultados no eran tan concluyentes, pero en términos generales **se puede apuntar a que el enfoque TDD facilita la producción de clases más sencillas y en ocasiones métodos también menos complejos**.

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 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:



Código 1. Ejemplo de prueba unitaria

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 ejecuten 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:

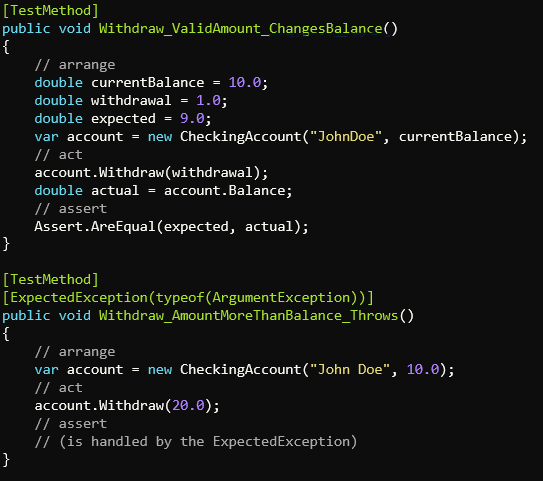
1. **Como programadores hemos realizado un esfuerzo de análisis**. Hemos decidido qué vamos a programar y qué no (el alcance del código).
2. También **hemos realizado un esfuerzo de diseño**. «TDD es una técnica de diseño lógico» (Beck, 2001, p. 88), puesto que al escribir el test estamos definiendo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

El patrón AAA

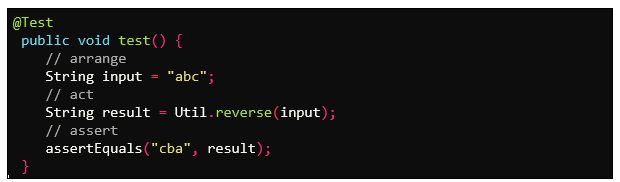
El patrón AAA, —*Arrange Act Assert* (Preparar, Actuar/Ejecutar, Afirmar/Comprobar)— es una manera habitual para definir las pruebas unitarias.

* 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.



Código 2. Dos ejemplos de prueba unitaria en C# siguiendo el patrón AAA. Fuente: Microsoft (2018)



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. En este curso no nos centramos en una herramienta o lenguaje específicos, pero es importante que consultes y conozcas los *frameworks* más habituales relacionados con la tecnología de desarrollo más afín a tus intereses.

El ***framework* Serenity,** por ejemplo, permite automatizar las pruebas en Java, integrándose también con herramientas BDD, como Cucumber o JBehave. Robot

Framework es el *framework* más popular 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).

Dedica algún tiempo a buscar en Internet aquellos *frameworks* y herramientas más próximos a tus intereses, y realiza algún tutorial sencillo para familiarizarte con su uso. Al final de este tema, en la sección «A fondo» encontrarás algunos recursos que podrán orientarte.

Como ejemplo, en el siguiente vídeo aprenderás a implementar un pequeño proyecto.

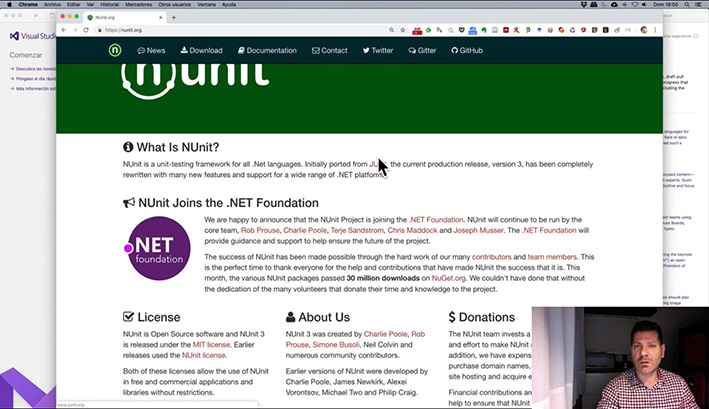
**Vídeo: TDD en Visual Studio (C# Y NUnit)**

En este ejemplo vamos a usar Visual Studio, el lenguaje C# y NUnit, un *framework* de *testing* compatible con todos los lenguajes .NET (NUnit, 2018).

En el vídeo vamos a implementar 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:

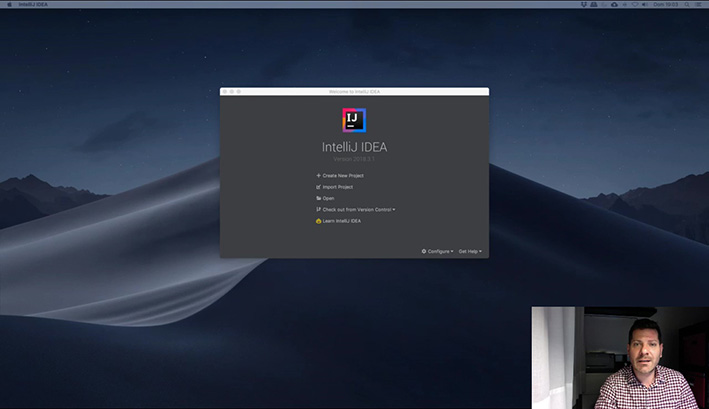
«Escribe un programa que muestre en orden los números del 1 al 100. Pero cuando el número sea múltiplo de 3 se mostrará la palabra "Fizz" en vez del número, y cuando sea múltiplo de 5 se mostrará la palabra "Buzz" en vez del número. Si el número es múltiplo de 3 y de 5 se mostrará la palabra "FizzBuzz".»



Accede al vídeo a través del aula virtual.

**Vídeo: TDD en Java (IDEA y JUnit)**

En este otro vídeo podrás ver el mismo ejemplo resuelto en lenguaje Java y empleando el *framework* de pruebas JUnit.



Accede al vídeo a través del aula virtual.

8.5. *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 unas 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 10, 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».

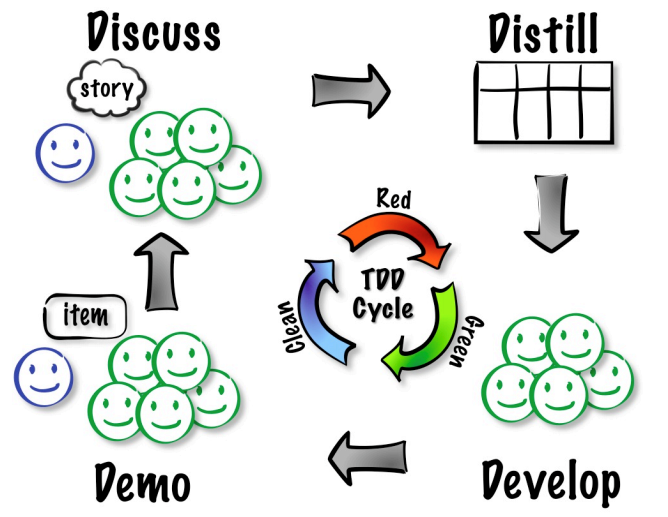


Figura 10. Modelo de ciclo de vida ATDD. Fuente: Hendrickson (2008)

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

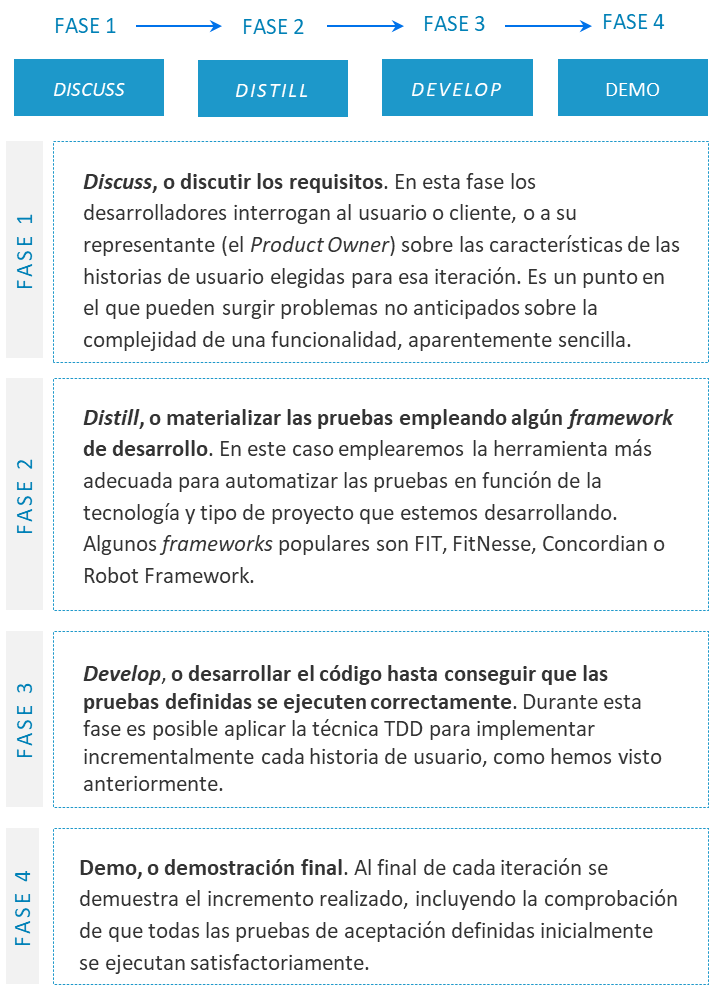


Figura 11. Fases de ATDD

8.6. *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, 2018), 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 los 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):

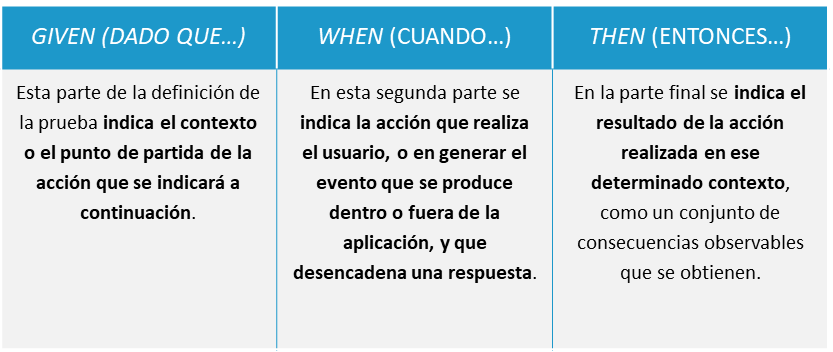


Figura 12. Partes de la notación GWT

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**.

8.7. BDD y Cucumber

En esta sección vamos a realizar una breve revisión del ***framework* Cucumber, que 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, & 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, como veremos.

En los **archivos Gherkin** (de extensión «.feature»), con una sintaxis próxima al lenguaje natural, se define el caso de prueba para un comportamiento dado. En él se define un conjunto de pasos, que en última instancia es un concepto análogo de invocación funcional (Cucumber Docs, 2018a).

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.

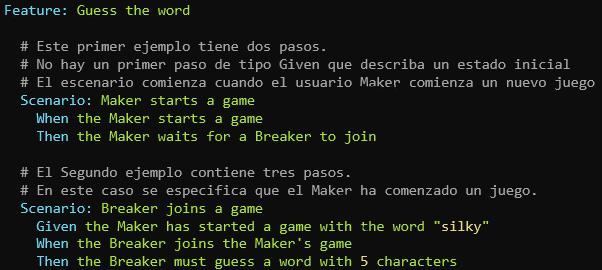
Introducción al lenguaje Gherkin

Como hemos visto, **Cucumber es una herramienta capaz de ejecutar de manera secuencial un conjunto de instrucciones definidas en un archivo de tipo «feature»**, o característica. Estos archivos **se escriben en un lenguaje llamado Gherkin**, que básicamente es texto plano en inglés, que resulta inteligible para cualquier lector. Este lenguaje de alto nivel permite que personas con diferentes perfiles, no necesariamente técnicos, puedan ponerse de acuerdo en la manera de definir las pruebas desde un punto de vista basado en el comportamiento esperado.

Para todo ello, Gherkin proporciona los siguientes elementos principales que se corresponden con ***keywords* o palabras reservadas del lenguaje** (Cucumber Docs, 2018b):

* ***Feature,* o característica.** Podemos considerarlas como unidades de funcionalidad aisladas dentro de un proyecto. Su propósito es el de proporcionar una descripción de alto nivel de una característica del sistema, o un conjunto de escenarios relacionados. Cualquier documento Gherkin comienza por la palabra *Feature* seguida de dos puntos y un texto breve que describe la característica (ver el ejemplo del Código 3). Cada característica agrupa un conjunto de escenarios o ejemplos.
* ***Example*** o ***Scenario*** (ambas palabras son sinónimos en Gherkin). Se trata de escenarios de utilización concretos, o ejemplos que ilustran una regla de negocio específica. Cada escenario agrupa un conjunto de *steps* o pasos, y cumple una triple misión en el contexto de BDD:
  + **Describe la especificación de una funcionalidad** del sistema, como si de un requisito se tratara.
  + **Es** **un elemento de documentación**, pues la propia sintaxis describe claramente el objetivo de ese fragmento de código.
  + **Define una prueba de validación** concreta del sistema.
* ***Steps***. Cada paso define una acción concreta dentro de la secuencia, o comportamiento, que representa cada escenario. Cucumber se encarga de leer los pasos definidos dentro de cada escenario de un archivo «.feature», y ejecutar el código correspondiente buscando en los *scripts* asociados.
* ***Given, When, Then, And, But*** se utilizan para definir *Steps* (pasos):
  + **Los pasos de tipo *Given*** describen el contexto inicial para un escenario. Generalmente representan acciones que se han producido en el pasado y que determinan un punto de partida para el ejemplo actual (por ejemplo, que el usuario ya ha iniciado sesión, o que hay determinada información contenida en la base de datos). La ejecución de código asociada se encarga de configurar el entorno de pruebas para replicar estas condiciones.
  + **Los pasos de tipo *When***permiten describir una acción concreta, ya sea producida por un usuario o por un evento desencadenado por otro sistema (por ejemplo, que el usuario pulsa un botón, o que se recibe determinada información). Es recomendable tener un único paso de tipo *When* en cada escenario y si aparecen más es síntoma de que ese ejemplo debería descomponerse en otros más sencillos.
  + **Los pasos de tipo *Then***indican un resultado esperado al final de la secuencia de pasos. El código correspondiente que implementa un paso de tipo *Then* debería contener una aserción para comparar el resultado real (lo que produce el sistema en su funcionamiento) con el resultado esperado (lo que el paso correspondiente indica que debería producirse en la definición del ejemplo o escenario).
  + **Las palabras *And y But*** se utilizan para dar mayor expresividad a la definición del escenario.

En el Código 4 se muestra un ejemplo de archivo «.feature» donde se utilizan las palabras reservadas *Feature, Scenario, When, Then,* y *Given,* junto con algunos comentarios comenzados por el símbolo #.

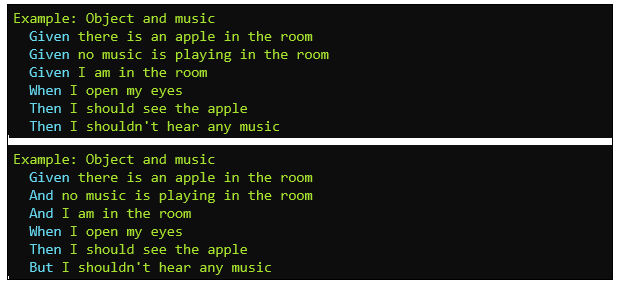


Código 4. Ejemplo de archivo «.feature» en lenguaje Gherkin. Fuente: Cucumber Docs (2018)

Como podemos ver, este ejemplo es fácil de interpretar. Describe una característica del producto, posiblemente un juego, que permite adivinar palabras. En un **primer ejemplo o escenario** se indica que un usuario denominado «Maker» crea un nuevo juego y se mantiene a la espera hasta que el «Breaker» se une.

El **segundo escenario** expresa la secuencia de acciones que indica el comienzo de un juego donde, además, se particulariza el ejemplo indicando la palabra elegida y el número de caracteres correspondiente.

Finalmente, en el Código 5 se muestra un ejemplo de utilización de las palabras ***And*** y ***But***, donde el mismo escenario se representa de manera equivalente, siendo mucho más expresiva y cercana al lenguaje natural la segunda implementación.



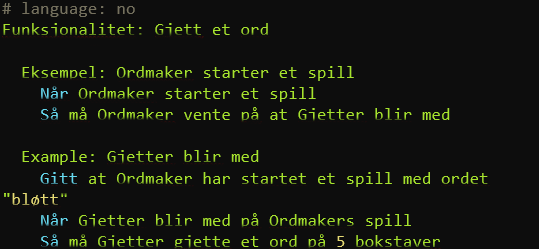
Código 5. Dos ejemplos equivalentes del mismo escenario, donde en el segundo se emplean las palabras reservadas ***And*** y ***But*** en el lenguaje Gherkin

Aunque lo habitual (y lo más recomendable) es trabajar siempre en lengua inglesa, existen traducciones de esta sintaxis a un total de 70 idiomas, incluido el español.

Accede al listado a través del aula virtual o desde la siguiente dirección:

[https://docs.cucumber.io/gherkin/reference/#spoken-languages](#spoken-languages)

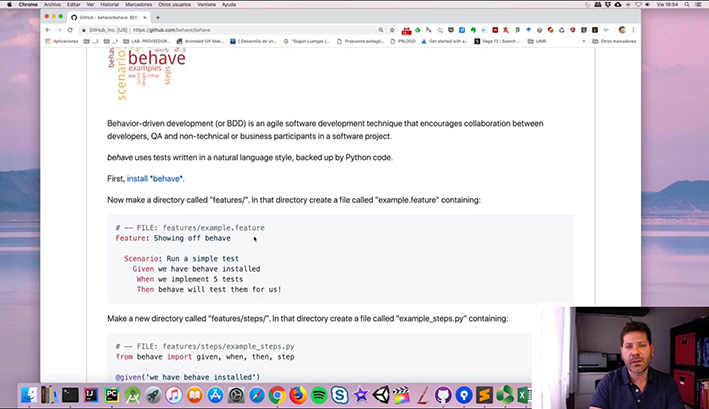
Y a continuación puedes ver un ejemplo en noruego:



Código 6. «Feature» en noruego

**Vídeo: BDD en Django**

En el siguiente videotutorial podrás ver un ejemplo de utilización de Cucumber y Python, empleando la técnica BDD con el *framework* de desarrollo Django.



Accede al vídeo a través del aula virtual.

8.8. Referencias bibliográficas

Agile Alliance (2015). TDD. En *Glossary.* Recuperado de <https://www.agilealliance.org/glossary/tdd/>

Agile Alliance. (2015). Acceptance Test Driven Development (ATDD). En *Glossary*. Recuperado de <https://www.agilealliance.org/glossary/atdd/>

Agile Alliance. (2015). BDD: Learn about Behavior Driven Development. En *Glossary*. Recuperado de <https://www.agilealliance.org/glossary/bdd/>

Agile Alliance. (2015). What is «Given - When - Then»? En *Glossary*. Recuperado de 2018, de <https://www.agilealliance.org/glossary/gwt/>

Beck, K. (2001). Aim, Fire. *IEEE Software, 18*(5), 87-89. <https://doi.org/10.1109/52.951502>

Boehm, B. W. (1981). *Software Engineering Economics* (1ª ed.). Englewood Cliffs, N.J: Prentice Hall.

Bolaños, D., Sierra, A. y Alarcón, M. I. (2007). *Pruebas de Software y JUnit. Un análisis en profundidad y ejemplos prácticos*. Madrid: Pearson Educación, S.A.

Cucumber Docs. (2018). Cucumber Reference [Web]. Recuperado de <https://docs.cucumber.io/cucumber/api/>

Cucumber Docs. (2018). Gherkin Reference [Web]. Recuperado de 2018, de <https://docs.cucumber.io/gherkin/reference/>

Fowler, M. (2005). Test Driven Development [Artículo Web]. Recuperado de <https://martinfowler.com/bliki/TestDrivenDevelopment.html>

Fowler, M. (2014). UnitTest [Artículo Web]. Recuperado de <https://martinfowler.com/bliki/UnitTest.html>

Gal-Oz, N., & Balaban, M. (2016). *Software testing: building acceptance tests.* [Material de clase del curso de fundamentos de ingeniería del software]. Universidad Ben-Gurion del Negrev (Israel), Be’er Sheva. Recuperado de <https://www.cs.bgu.ac.il/~fsen161/Class_Material>

Ghory, I. (2007). Using FizzBuzz to Find Developers who Grok Coding [Entrada de un blog]. Recuperado de de <https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/>

Grigg, J. (2012). Arrange Act Assert [Wiki]. Recuperado de <http://wiki.c2.com/?ArrangeActAssert>

Hartikainen, J. (2015). What’s the difference between Unit Testing, TDD and BDD? [Blog personal]. Recuperado de <https://codeutopia.net/blog/2015/03/01/unit-testing-tdd-and-bdd/>

Henderson-Sellers, B. (1995). *Object-Oriented Metrics: Measures of Complexity* (1.a ed.). Upper Saddle River, N.J: Prentice Hall.

Hendrickson, E. (2008). Driving Development with Tests: ATDD and TDD. Quality Tree Software, Inc. Recuperado de <http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf>

Janzen, D. S., Polytechnic, C., Obispo, S. L., & Saiedian, H. (2008). Does Test-Driven Development Really Improve Software Design Quality. *IEEE Software*, Marzo/Abril 2008, 77–84.

JBehave. (2018). What is JBehave? [Página de oficial del producto]. Recuperado de <https://jbehave.org/>

Microsoft. (2018). Unit Test Basics in Visual Studio [Web]. Recuperado de <https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics>

Murphy, C. (2005). Improving Application Quality Using Test-Driven Development (TDD). *Methods and Tools, a Software Development Magazine*, Spring’05. Recuperado de <http://www.methodsandtools.com/archive/archive.php?id=20>

North, D. (2006, septiembre 20). Introducing BDD [Artículo Web]. Recuperado de <https://dannorth.net/introducing-bdd/>

North, D. (2009). How to sell BDD to the business [Vídeo]. London. De <https://skillsmatter.com/skillscasts/923-how-to-sell-bdd-to-the-business>

NUnit. (2018). NUnit.org [Web]. Recuperado de <https://nunit.org/>

Pablos, J. (2 de mayo de 2018). Top 7 Frameworks de Pruebas Automatizadas [Entrada de blog]. Recuperado de <https://josepablosarco.wordpress.com/2018/05/02/top-7-frameworks-de-pruebas-automatizadas/>

Pressman, R. S. (2010). *Ingeniería del software: un enfoque práctico* (7.a ed.). México: McGraw-Hill Educación.

Rose, S., Wynne, M., & Hellesøy, A. (2015). *The Cucumber for Java Book*. Dallas, Texas: Pragmatic Programmers.

Sharma, L. (2017). Difference between Verification and Validation [Tutorial]. Recuperado de <http://toolsqa.com/software-testing/difference-between-verification-and-validation/>

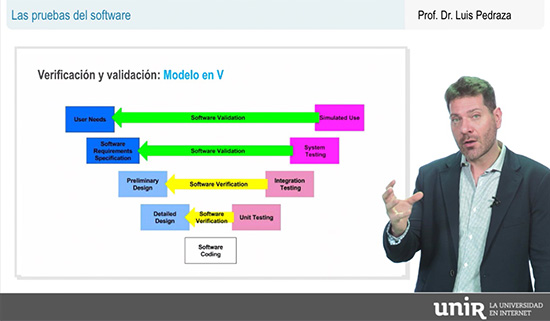
STF. (2011). Verification vs Validation [Web]. Recuperado de <http://softwaretestingfundamentals.com/verification-vs-validation/>

WikiWikiWeb. (2014). Fizz Buzz Test. Recuperado de <http://wiki.c2.com/?FizzBuzzTest>

A fondo

Lección magistral: Las pruebas del software

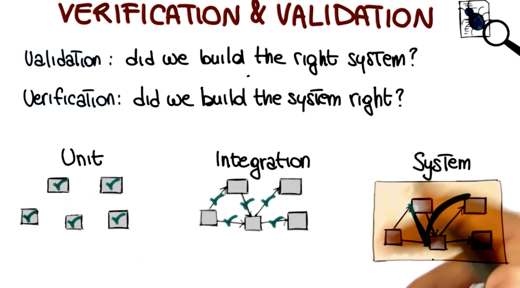
En esta lección magistral se presentan dos conceptos fundamentales en ingeniería del software: la validación (que permite garantizar que el producto a construir es correcto, y cumple con las expectativas del cliente) y la verificación (que está más relacionada con garantizar que el proceso de construcción del producto sea correcto). En la lección también se presentan los diferentes tipos de pruebas que podemos encontrar en proyectos de desarrollo.



Accede a la lección magistral a través del aula virtual.

Verification & Validation

En este vídeo del Georgia Tech, Alex Orso explica los conceptos de verificación y validación del software, junto con los diferentes niveles en los que puede realizarse el proceso de verificación. El vídeo pertenece a un curso específico sobre el proceso de desarrollo de software ofrecido por Udacity (las lecciones 13, 14 y 15 tratan aspectos relacionados con las pruebas del software).



Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

https://youtu.be/gQrSxbfUjug

Accede al curso desde la siguiente dirección web:

https://classroom.udacity.com/courses/ud805

Diccionario de pruebas

En la siguiente web encontrarás un completo diccionario de términos relacionados con el ámbito de las pruebas de software. Encontrarás definidos los conceptos de verificación y validación, pero también muchos conceptos relacionados con prácticas concretas de pruebas que permiten valorar la idoneidad del sistema desarrollado.



Accede a la página web a través del aula virtual o desde la siguiente dirección web:

https://www.tutorialspoint.com/software\_testing\_dictionary/

Cucumber Tutorial

En la siguiente web encontrarás un tutorial de Cucumber, con ejemplo de utilización de esta potente herramienta en lenguaje Java y en lenguaje Ruby. Se incluye una descripción de los principales elementos de la sintaxis del lenguaje Gherkin.



Accede a la página web a través del aula virtual o desde la siguiente dirección web:

https://www.tutorialspoint.com/cucumber/

*Does Test Driven Development Work?*

Este artículo resulta interesante, no solo porque el autor se muestra algo escéptico con los beneficios que la práctica del *Test Driven Development* supone, sino por la cantidad de recursos e información que utiliza como referencia.



En él encontrarás enlaces a artículos de grandes gurús del desarrollo, como Kent Beck (padre de la metodología ágil XP), David Heinemeier Hansson (creador de Ruby on Rails) o Bob Martin —«Uncle Bob», uno de los firmantes del Manifiesto Ágil y autor de varios libros sobre ingeniería del software en un contexto ágil—. Especialmente interesante es una secuencia de 4 videoconferencias entre Kent Beck, MartinFowler y David Heinemeir Hansson (DHH) sobre el futuro de TDD como práctica.

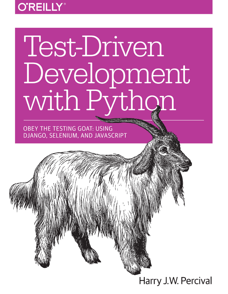
Accede al documento a través del aula virtual o desde la siguiente dirección web:

<https://dev.to/ruairitobrien/does-test-driven-development-work-p54>

*Test–Driven Development with Python*

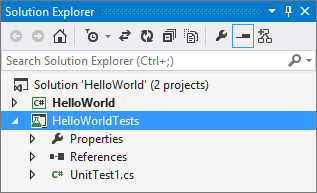
Percival, H. (2014). *Test-driven development with Python* (First edition). Sebastopol, CA: O’Reilly Media.

Este libro trata en profundidad las técnicas de desarrollo dirigido por pruebas para aplicaciones web, empleando para ello el lenguaje Python y el *framework* de desarrollo Django. Además, el autor pone a disposición de los lectores una versión *online* accesible de manera completamente gratuita.



Testing tools in Visual Studio

En esta sección de la documentación oficial de Visual Studio encontrarás una introducción a la manera de realizar pruebas en este entorno. Especialmente interesante es la sección sobre pruebas unitarias, donde se describe el **Test Explorer**, el *framework* de pruebas unitarias y otras herramientas, como **Microsoft Fakes** que permite simular la integración con otros componentes para poder centrar las pruebas en una unidad concreta de código.

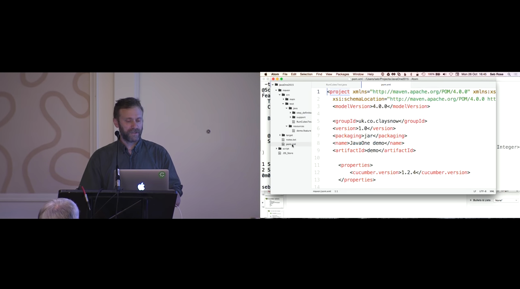


Accede al documento desde la siguiente dirección web:

https://docs.microsoft.com/en-us/visualstudio/test/improve-code-quality

An introduction to Behavior-Driven-Development (BDD) with Cucumber for Java

En este vídeo encontrarás una completa introducción al desarrollo dirigido por pruebas en Java empleando el *framework* Cucumber. El instructor, Seb Rose, es autor del libro *The Cucumber for Java Book*, y publica artículos relacionados con la ingeniería del software, el desarrollo ágil, y el *testing*, en el blog de su consultora tecnológica: Claysnow.



Accede al vídeo desde la siguiente dirección web:

<https://dev.to/ruairitobrien/does-test-driven-development-work-p54>

Actividades

Laboratorio: Desarrollo dirigido por pruebas en un proyecto sencillo

En este trabajo deberás **implementar una aplicación sencilla** empleando una de las técnicas de desarrollo estudiadas en este tema: **TDD o BDD**. Para ello, tendrás completa libertad a la hora de elegir la tecnología, lenguaje y *framework* de desarrollo más apropiados según tu experiencia e intereses.

La aplicación consistirá en una calculadora sencilla capaz de realizar las siguientes operaciones:

* **Suma y resta de números enteros**.
* **Multiplicación y división de números enteros**.
* **Raíz cuadrada de números enteros**.

Para implementarla no podrás hacer uso de ninguna librería matemática y deberás utilizar operaciones básicas como la suma, resta, multiplicación y división, empleando para ello las funciones de la propia calculadora implementadas en los dos pasos anteriores. La precisión alcanzada en los cálculos debe ser superior a 10-5.

Objetivos de la actividad

Con esta actividad pretendemos alcanzar los siguientes objetivos:

* **Desarrollar un producto web sencillo** utilizando los principios del desarrollo dirigido por pruebas.
* **Comprender la mecánica de trabajo** con la filosofía del desarrollo dirigido por pruebas.
* Aprender a utilizar los **fundamentos de un sistema de control de versiones**, pues deberás publicar el código en un repositorio público *online*.
* **Aprender a buscar en Internet** información de calidad y específica para resolver un problema concreto.

Descripción de la actividad

No existe ninguna restricción en cuanto a la tecnología o formato de la aplicación. Puede ser una simple aplicación de línea de comandos en C# o en Java, una aplicación con GUI, una aplicación web, una aplicación móvil Android, una API… Encontrarás tutoriales específicos relacionados con las técnicas TDD y BDD para casi cualquier lenguaje y *frameworks* de desarrollo moderno existentes.

La actividad se trata por tanto de encontrar un tutorial, manual o artículo, relacionado con la tecnología más interesante para ti. Puede que el tutorial utilice como ejemplo específico el desarrollo de una calculadora, pero en cualquier caso te será fácil adaptar las indicaciones a una aplicación como la propuesta.

Basándote en el tutorial, deberás resolver la actividad implementando de manera incremental y secuencial las funcionalidades de suma, resta, multiplicación, división y raíz cuadrada, anteriormente descritas, y teniendo en cuenta las limitaciones impuestas.

Durante el trabajo de desarrollo deberás utilizar un **sistema de control de versiones Git**. Se trata de que dejes reflejada cualquier modificación mediante un **commit** en la rama master del repositorio, que será la única en la que debes trabajar.

Es importante que en la secuencia de commits quede claramente reflejado que defines la prueba correspondiente a cada funcionalidad antes de su implementación. Cualquier actividad de refactorización del código también debería quedar reflejada en la secuencia de commits.

Finalmente, deberás publicar el código en un repositorio *online*, como GitHub o Bitbucket (también de tu elección), no siendo necesario el envío de código con la actividad, sino únicamente un enlace al repositorio que contiene el código con la actividad resuelta según la técnica propuesta.

Entrega

Cada alumno entregará de manera individual un documento en formato PDF que deberá contener los siguientes apartados:

1. **Introducción**. Aquí deberás describir el proyecto indicando la tecnología, lenguaje o *framework* de desarrollo elegidos. También deberás indicar qué fuentes —documentales, tutoriales, vídeos o libros— has utilizado para resolver la actividad, referenciando cada uno de ellos como se hace en los apuntes de esta asignatura (empleando la normativa de referencias bibliográficas APA o IEEE). Deberás indicar de manera justificada por qué has elegido ese documento y de qué manera lo has adaptado para resolver la actividad.

Extensión aproximada: 2 páginas.

1. **Código de las pruebas**. En esta sección deberás copiar el código fuente de las pruebas que hayas definido para resolver la actividad.
2. **Código de la aplicación**. En esta sección deberías copiar el código fuente que resuelve la actividad propuesta, ya sea una aplicación o una librería.

**Además, es importante que en la portada del trabajo incluyas un enlace al repositorio Git que contiene el código fuente del proyecto.**

Nota: Para la inserción del código en el trabajo se recomienda utilizar un formateador de código fuente del estilo de «**hilitme»**.

Accede a la página web a través del aula virtual o desde la siguiente dirección:

<http://hilite.me/>.

Criterios de evaluación

Al evaluar esta actividad se tendrán en cuenta los siguientes aspectos:

* Correcta estructura y redacción del documento, sin faltas ortográficas y con capacidad de síntesis.
* Estética del documento, cuidando los aspectos visuales del texto y código incluidos en el trabajo.
* Correcta aplicación de la técnica TDD o BDD.
* Correcto manejo de la herramienta Git para el control de versiones.

Test

1. ¿De qué se encargan las técnicas de validación del software?

A. De comprobar la correcta especificación de requisitos según las necesidades del usuario.

B. De garantizar que el sistema final cumple con las expectativas del cliente o usuario.

C. De responder a la pregunta «¿estamos construyendo el sistema correcto?»

D. Todas las respuestas anteriores son correctas.

1. ¿Cuál es el objetivo principal de las pruebas del sistema?

A. Comprobar la corrección funcional de las capacidades implementadas.

B. Comprobar el correcto funcionamiento del sistema en un entorno operativo real, con especial atención a los requisitos no funcionales.

C. Garantizar la adecuación del sistema a las necesidades del cliente.

D. Comprobar la correcta interacción entre componentes del sistema en el entorno de desarrollo.

1. ¿Cuál de las siguientes pruebas se parece más a una prueba de validación?

A. Una prueba de validación de tipo alfa.

B. Una prueba unitaria.

C. Una prueba de validación de tipo beta.

D. Una prueba de integración entre todos los componentes del sistema.

1. ¿Cuál de las siguientes es una ventaja del enfoque de desarrollo TDD?

A. Ayuda a construir código que se comprueba de manera automática.

B. Aumenta la productividad de los equipos de desarrollo.

C. Ayuda a mejorar el diseño del producto, aunque no haya una fase específica dedicada a ello.

D. Las respuestas A y C son correctas.

1. ¿El concepto TDD es sinónimo de…?

A. Pruebas automatizadas.

B. *Test–first*.

C. BDD.

D. ATDD.

1. ¿En qué consiste el patrón AAA dentro de la técnica TDD?

A. En escribir los test teniendo en cuenta los intereses del usuario final.

B. En definir las pruebas teniendo en cuenta las acciones esperadas por parte del usuario.

C. En diseñar cada nuevo módulo antes de proceder a su implementación.

D. En estructurar las pruebas comenzando por la definición de parámetros de entrada, colocar luego las acciones a ejecutar y, finalmente, comprobar los resultados esperados.

1. ¿Cuál de los siguientes emparejamientos agrupa palabras que podemos considerar conceptualmente similares dentro de los enfoques TDD y BDD?

A. *Act* y *When*, *Arrange* y *Given*, *Assert* y *Then*.

B. *Act* y *When, Assert* y *Given, Arrange* y *Then*.

C. *Act* y *Then, Arrange* y *Given, Assert* y *When.*

D. Ninguna de las respuestas anteriores es correcta.

1. ¿Qué beneficios aporta la técnica de desarrollo ATDD sobre TDD?

A. Garantiza que el sistema no será finalizado hasta que se verifiquen todas las pruebas de aceptación.

B. Ayuda a modelar las pruebas desde el punto de vista de los resultados esperados por el cliente.

C. Involucra en el proceso de desarrollo y pruebas a un número mayor de *stakeholders*.

D. Las respuestas B y C son correctas.

1. ¿Qué palabras proporciona el lenguaje Gherkin para definir la secuencia de pasos de una funcionalidad?

A. *Step, Given, When, Feature.*

B. *Feature, Given, When, Then.*

C. *And, But, Example, Feature.*

D. *Given, When, Then, And, But.*

1. Cuando empleamos un enfoque de desarrollo dirigido por las pruebas, esto significa que…

A. Debemos definir todas las pruebas al comienzo del desarrollo del sistema.

B. Debemos escribir las pruebas inmediatamente después de finalizar la fase de diseño.

C. Debemos emplear obligatoriamente un *framework* apropiado que ayude a automatizar las pruebas.

D. Ninguna de las respuestas anteriores es correcta.