RIC REVISTA DE INICIACIÓN CIENTÍFICA
ISSN-L: 2413-6786
Revista de Iniciación Científica, Vol. 8, No. 2, pp. 50 – 56, julio – diciembre 2022
Portal de revistas: http://revistas.utp.ac.pa

Prácticas orientadas por pruebas para el desarrollo de *software*, una revisión sistemática

Test-oriented practices for software development, a systematic review

Raúl I. Ramírez B. [1], *Angela R. Pellecchia M.*[1], *Ana G. Saa Morales*[1], *Analissa Santos Quiel*[1], *Holger De J. González M.*[1], *Belén Bonilla-Morales*[2,*]

[1]Universidad Tecnológica de Panamá, Facultad de Ingeniería de Sistemas Computacionales, Panamá
[2]Universidad Tecnológica de Panamá, Facultad de Ingeniería de Sistemas Computacionales, Departamento de Ingeniería de Software, Panamá

Fecha de recepción: 8 de febrero de 2022. **Fecha de aceptación:** 15 de mayo de 2022.
***Autor de correspondencia:** belen.bonilla@utp.ac.pa

Resumen. En la actualidad, se busca que el proceso de desarrollo de *software* sea más rápido y efectivo. Las principales causas que impiden cumplir con este objetivo son los errores en las especificaciones de requisitos y la ausencia de pruebas acordes al producto construido. Como consecuencia, se obtienen productos de *software* que no alcanzan la calidad esperada por los clientes. Por ello, es crítico que los equipos de desarrollo de *software* le den prioridad a la etapa de pruebas, así como a la definición correcta de los requisitos, para evitar crear un producto que pueda resultar en la pérdida de recursos y en la insatisfacción de los *stakeholders*. Como respuesta a esta problemática, en el ámbito de desarrollo de *software*, se han diseñado distintas prácticas basadas en pruebas para minimizar errores en el proceso de desarrollo, entre estas, se pueden mencionar: *Test-Driven Development, Behaviour-Driven Development* y *Acceptance Test Driven Development*. En este artículo, se aborda una revisión literaria de estas prácticas con el objetivo de determinar la importancia de cada una, en el proceso de desarrollo de *software*, mediante el análisis de sus características y diferencias.

Palabras clave. ATDD, BDD, desarrollo de *software*, lenguaje ubicuo, pruebas de aceptación, pruebas de *software*, TDD.

Abstract. Nowadays, the intent is to make the software development process faster and more effective; However, the main causes that prevents meeting this goal, are errors with the software requirements specifications and the absence of the necessary testing for the product that has been build. Thereby the resulting software doesn't reach the quality expected by the clients. Therefore, development teams must prioritize the testing stage and a correct definition of the requirements to avoid having an unstable product that can result in the loss of resources and dissatisfaction of stakeholders. In response to this problem, in the software field, different test-based practices have been designed to minimize errors in the development process. Among these practices are Test-Driven Development, Behavior-Driven Development, and Acceptance Test-Driven Development. Throughout this paper, we have conducted a literary review of these practices in the software development process, by analyzing their characteristics and differences.

Keywords. ATDD, BDD, software development, ubiquitous language, acceptance testing, software testing, TDD.

1. Introducción

De acuerdo con Frederick Brooks "la parte difícil de desarrollar *software* es la especificación, el diseño y las pruebas" [1]. Desde los inicios de la Ingeniería de *Software* ha existido lo que se conoce como la "crisis del *software*" [2] que, a resumidas cuentas, implicaba que el *software* que se construye no satisface los requerimientos ni las necesidades del cliente y su desarrollo excede los recursos disponibles para tal fin.

Hoy en día, uno de los principales problemas en el desarrollo de *software* es que la mayoría de los proyectos, desde que son definidos, inician directamente con la etapa de programación, y en algunos es la única etapa que se lleva a cabo.

https://doi.org/10.33412/rev-ric.v8.2.3672

En las últimas décadas, se han creado prácticas o técnicas para tratar de solventar o evitar la famosa crisis del *software*.

En este artículo, se examinarán a profundidad algunas de las prácticas orientadas por pruebas para el desarrollo de *software*. En la siguiente sección, se presenta la metodología utilizada para realizar la investigación. En la sección 3, se encuentran los resultados que responden a las preguntas de investigación contempladas en la metodología. En la sección 4, se plantea una discusión sobre las diferencias entre las metodologías investigadas. Finalmente, se concluye el artículo y se presentan recomendaciones para futuras revisiones literarias.

2. Metodología

Para realizar esta investigación se siguió un proceso que contempló tres etapas: planificación, revisión y reporte.

2.1 Planificación de la revisión literaria

Durante la etapa de planificación, se confirmó que los requerimientos de la investigación eran propicios para realizar una revisión sistemática de la literatura. Luego, se utilizaron los requerimientos para plantear las siguientes preguntas de investigación para las prácticas de *Test-Driven Development*, *Behavior-Driven Development* y *Acceptance Test-Driven Development*:

a. ¿En qué consiste la práctica?
b. ¿Cuáles son las características de la práctica?
c. ¿Cómo es el proceso que involucra esta práctica?
d. ¿Qué herramientas tecnológicas le dan soporte a esta práctica?

Luego de tener las preguntas de investigación, se desarrolló un protocolo de revisión que establecía como fuentes de información artículos, libros, tesis, documentos oficiales, bitácoras de eventos, revistas documentales con hasta 10 años de antigüedad; y entradas de blogs o sitios web escritos por profesionales idóneos con hasta 5 años de antigüedad; en los idiomas español o inglés.

2.2 Revisión literaria

En la etapa de revisión literaria, se efectuó una búsqueda general para determinar qué literatura estaba disponible en la web. Luego, se identificaron las fuentes que se seleccionarían para llevar a cabo la revisión, estas debían cumplir con los requerimientos establecidos en el protocolo de revisión. Posteriormente, se procedió a sintetizar la información obtenida y se realizó una revisión de calidad de esta síntesis para asegurar que no contaba con plagio, y tenía coherencia y cohesión.

3. Desarrollo y discusión

En esta sección se plantean los resultados de la revisión bibliográfica, específicamente, la solución a las preguntas planteadas en la metodología; seccionada en tres, donde cada sección corresponde a las prácticas que fueron sujeto de estudio en esta investigación.

3.1 *Test-driven development*

Test Driven Development (TDD), es un método de diseño y programación presentado por Kent Beck, como parte de la metodología ágil *Xtreme Programming* [1]. TDD se basa en comenzar creando pruebas que fallen, para posteriormente empezar a desarrollar el código. Estas pruebas se aplican a una parte o unidad del código, es decir, son pruebas unitarias.

Entre las características que hacen al TDD una práctica beneficiosa están:

- Permite desarrollar un código mejor diseñado, más limpio y escalable.
- Facilita la codificación y mejora la comunicación en el equipo de desarrollo, debido a que los casos de prueba son el primer paso de la práctica.
- Ayuda a encontrar el origen del defecto, gracias a la documentación de los casos de prueba.

3.1.1 Subprácticas del TDD

Esta metodología incluye tres subprácticas, las cuales son [2]:

Test-first: consiste en escribir las pruebas antes de pasar al desarrollo del código, esto lo hace el mismo programador, con el objetivo de tener una comprensión de lo que se desea desarrollar.

Automatización: las pruebas se tienen que escribir en código para poder así ejecutarlas cuando sea requerido, y de esta manera, poder saber si lo que se está probando funciona de la manera correcta o no.

Refactorización posterior: se reestructura el código sin cambiar la funcionalidad, buscando así, poder mantener la calidad de la arquitectura, y hacer más comprensible y claro el código para facilitar el posterior mantenimiento.

3.1.2 Proceso del TDD

En la figura 1, se puede apreciar el funcionamiento del ciclo de vida de TDD, iniciando con la selección de un pequeño incremento de funcionalidad, se escribe el caso de prueba para dicha tarea, produciéndose una prueba que falla. Posterior a eso, se hace la codificación y se vuelven a correr las pruebas, si estas fallan, se regresa a la codificación para corregirla, y se

vuelven a ejecutar las pruebas. Si la prueba pasa, entonces se procede a la refactorización del código [3].

3.1.3 Herramientas tecnológicas utilizadas en TDD

Algunas de las herramientas tecnológicas que dan soporte a las práctica de TDD son: *csUnit* [4] y *NUnit* [5], para proyectos .NET; *DocTest* [6] y *PyUnit* [7], para proyectos en Python; *JUnit* [8] y *TestNG* [9], para Java; *PHPUnit* [10], para proyectos en PHP; *RSpec* [11], para proyectos en Ruby.

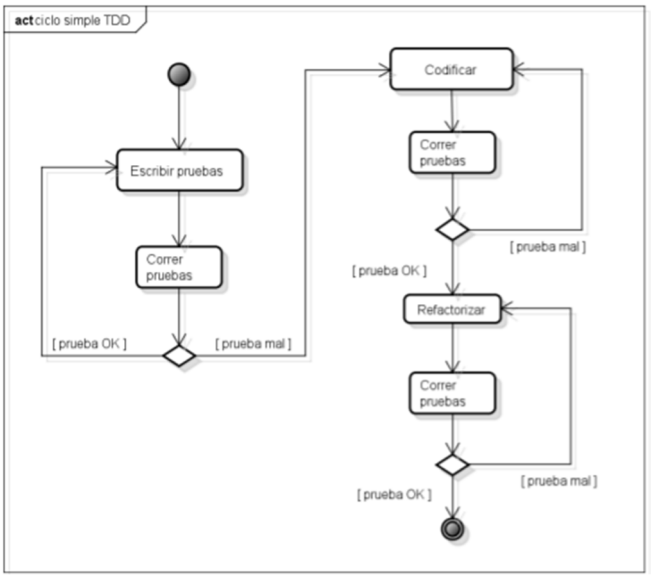

Figura 1. Diagrama de actividades del ciclo de TDD [2].

3.2 *Behavior-driven development*

Al momento de desarrollar un proyecto de *software*, generalmente, ocurren discrepancias entre los *stakeholders*, el equipo de desarrollo y *testers*, debido a que se tiende a que la comunicación sea mediante diferentes niveles de lenguajes. A causa de esto, se puede perder información o incluso malinterpretar cuáles son realmente las necesidades de los clientes para el *software* a realizar, resultando, en un *software* que no cumple con los verdaderos requisitos de los clientes, pérdida de recursos y, sobre todo, la insatisfacción de los clientes.

A razón de lo mencionado anteriormente, se crea el proceso de *Behavior –Driven Development*, conocido también como BDD, creado por el inglés Dan North [12]. Este es un conjunto de prácticas de la ingeniería de *software* diseñadas a partir del proceso *Test Driven Development* (TDD), con el objetivo principal de que los equipos de desarrollo de *software* realicen sistemas más valiosos, con mejor calidad y, sobre todo, que cumplan con las necesidades del usuario y los *stakeholders*, mediante la comprensión del comportamiento (*behaviour*) del sistema.

En la práctica del BDD, se hace hincapié a la buena comunicación entre las partes para entender los requerimientos a diferencia de un modo de desarrollo tradicional, donde en cada cambio de fase, se trabaja con la información que indicó el equipo anterior, dando espacio a que exista pérdida de información o equivocaciones al cambiar de nivel de lenguaje.

En este método de desarrollo se unen los equipos, desde *stakeholders* hasta *testers,* para definir los requerimientos de manera que queden claros para todos.

3.2.1 Procesos y prácticas de BDD

El enfoque de BDD se puede agrupar de manera macro en dos grupos. El primero, es donde se establecen ejemplos mediante lenguajes ubicuos para demostrar el comportamiento del *software.* Dichos ejemplos, a su vez, se convierten en historias de usuarios o requerimientos.

El segundo grupo, se basa en utilizar los ejemplos establecidos en el primer enfoque, como base de pruebas automatizadas. De igual manera, se verifican las funcionalidades establecidas para el usuario. Esto asegura, que el sistema se comporte de acuerdo con los requisitos del negocio.

El proceso de BDD, indicado en la figura 2, se desarrolla con una cadena de prácticas, iniciando con la identificación de los objetivos del negocio. Posteriormente, se buscan las características que permitan cumplir con esos objetivos, estos se definen de la mano del equipo. Para ilustrar mejor se crean ejemplos, que luego se automatizan en especificaciones ejecutables que permiten validar si el *software* que se crea se comporta de manera esperada [13].

Figura 2. Diagrama del conjunto de prácticas que se obtienen al seguir el proceso de desarrollo BDD.

3.2.2 Uso de lenguaje ubicuo

Lo peculiar de la práctica de BDD, es el uso de un lenguaje ubicuo; técnica acogida del libro *Domain-Driven Design* [14], para que el vocabulario que se va a manejar entre el equipo de desarrollo y el negocio sea persistente y claro.

Puntualmente, el lenguaje ubicuo es utilizado en el proceso de análisis como tal. Por ello, se establecieron patrones que se utilizan para crear ejemplos, y estos a su vez, se convierten en requerimientos o historias de usuarios.

Se diseñaron dos patrones, "*Given-When-Then*" y "*Role-Feature-Reason*" cuyo patrón *es* "*As a -I want -So that*". En español, se traducen como "Dado-Cuando-Entonces" y "Como-Deseo-Para que" respectivamente. A partir de estos patrones, las historias o ejemplos creados, se capturan los criterios de aceptación. Si el sistema satisface el criterio de aceptación, entonces este se comporta correctamente; si no es el caso, el sistema no se comporta como debería hacerlo.

Para ejemplificar lo mencionado anteriormente, se diseñó el ejemplo presentado en la figura 3, en base a la suscripción de una plataforma de películas online.

Característica: Crear subscripción a plataforma de películas
Como Nuevo cliente de X compañía
Deseo crear una suscripción al plan estándar
Para ver películas con opción a una pantalla.

Escenario: Crear subscripción a plan estándar
Dado un plan escogido
Cuando cree una nueva cuenta
Y haya pagado la suscripción del plan escogido
Y haya aceptado los términos
Entonces podrá visualizar películas.

Figura 3. Ejemplo de patrones utilizados para control de vocabulario.

3.2.3 Herramientas tecnológicas utilizadas en BDD

En cuanto a herramientas que se pueden utilizar para la práctica de BDD, se pueden encontrar herramientas de *software* como *frameworks* de lenguajes de programación. Ejemplo de ellos son *Jbehave* [15] , *Instinct* [16], y *beanSpec* [17], basados todos en Java y *PHPSpec* [18], y *Behat* [19], para lenguajes PHP. Por otro lado, en la actualidad se utiliza frecuentemente la herramienta *Cucumber* [20]. Esta utiliza la sintaxis *Gherkin* y se puede utilizar como *framework* para pruebas automatizadas.

3.3 *Acceptance test-driven development*

El desarrollo basado en pruebas de aceptación (ATDD), es una técnica de desarrollo de *software* que combina la especificación de requisitos con pruebas ejecutables automáticas de estos requisitos. Existen diferentes marcos de pruebas de aceptación que están divididas de la siguiente manera: basados en tablas, basados en texto y en lenguaje *scripting*. ATDD, tiene como objetivo reunir a los ingenieros, *testers*, y desarrolladores de *software*, para poder lograr un entendimiento mutuo en diferentes perspectivas, lo que ayuda a prevenir errores y malentendidos entre *stakeholders*, llevando a producir programas que satisfagan los requerimientos reales, haciéndolos más simples y sencillos [21].

3.3.1 Características principales

- Comunicación y visión compartida. En vez de escribir requerimientos abstractos y detallados, en ATDD, la funcionalidad se detalla mediante la escritura temprana de pruebas de aceptación. Se inicia discutiendo y definiendo ejemplos de comportamiento concreto con el cliente y los más significativos se refinan y se agregan a las pruebas del sistema, estos juegan un papel central en la comunicación del equipo y el cliente, ya que definen lo que se va a construir.
- Guía del desarrollo: previo a la programación, las pruebas sirven para resolver dudas y ambigüedades de manera tal, que plantean un objetivo claro a cumplir por los programadores.
- Calidad en el proceso: Ayuda a prevenir los defectos durante el desarrollo y no solo al final.
- Integración con otras prácticas: ATDD, se puede utilizar con historias de usuario, casos de uso, etc., [22].

3.3.2 Proceso de ATDD

ATDD, es una técnica dirigida por pruebas, por ende, el primer paso es implementar una prueba. Pueden presentarse ocasiones donde no existe nada que probar como primera instancia, por lo que, es beneficioso y recomendado escribir una descripción sobre la nueva prueba, esta descripción es llamada "historia de usuario" [23].

La historia de usuario es una descripción breve y fácil de entender, de lo que la prueba de aceptación va a evaluar, tiene como propósito, captar un significado esencial de una prueba de aceptación y evitar que esta prueba sea repetida una cantidad innecesaria de veces. Estas historias de usuario están pensadas para que los expertos del dominio las entiendan fácilmente y sin ambigüedades, por lo que se escriben en lenguaje humano natural, aunque deben ser escritas por expertos (programadores), para lograr el máximo beneficio de esta técnica [24].

Después de que se diseña la prueba de aceptación, basada en una historia de usuario, se procede a implementarla. Las pruebas de aceptación a menudo buscan poner el sistema bajo prueba, en un determinado estado, para realizar debidas acciones en el sistema y comparar los resultados devueltos por el mismo, con los resultados esperados. Se considera que una prueba de aceptación falla cuando los resultados reales del sistema difieren de los resultados esperados, y es superada en caso contrario, cuando los resultados reales son iguales a los resultados esperados.

Con el resultado de la prueba se tiene información suficiente sobre lo que se debe hacer a continuación. En caso de que la prueba falle, se tendría que replantear la situación y hacer que la prueba pase. Cuando la prueba finalmente pasa, el diseño interno del *software* en construcción tendrá que ser revisado y mejorado. La prueba se encarga de que la función que se está construyendo no se rompa, cada modificación realizada debe someterse a una nueva ejecución de la prueba. Las demás pruebas pueden verse afectadas por las modificaciones, por lo que se deben volver a ejecutar. Una vez que el programador esté de acuerdo con el diseño, la función se puede considerar realizada y seguir con la próxima historia de usuario, esto hace que el flujo de trabajo sea cíclico e incremental [25].

3.3.3 Herramientas tecnológicas usadas en ATDD

En esta práctica se usan *softwares* de pruebas como *JUnit, Selenium* [26], *Cucumber*, *Fitnesee* [27], *Concordion* [28], *robot framework* [29], entre otras.

3.4 Discusión

En la tabla 1, se realizó una comparación de las tres prácticas orientadas por pruebas para el desarrollo de *software*, donde se establecieron como características comparativas: el objetivo de la práctica, su enfoque principal, el lenguaje utilizado para emplear o desarrollar la práctica, el nivel de pruebas en el que normalmente se enfoca cada práctica, los participantes que intervienen en el proceso de cada práctica, los tipos de proyectos en los que se usan y las herramientas tecnológicas que dan soporte a estas prácticas.

Tabla 1. Comparación entre TDD, BDD, ATDD

Característica	TDD	BDD	ATDD
Objetivo	Implementación de una característica.	Buscar el comportamiento correcto del sistema mejorando la colaboración y comunicación entre desarrolladores y stakeholders.	Cumplir con los requisitos del usuario.
Enfoque principal	Pruebas unitarias.	Demostrar el comportamiento del *software*.	Pruebas de aceptación.
Lenguaje utilizado	Lenguaje de programación.	Lenguaje natural.	Lenguaje natural.
Nivel de pruebas	Unitario, Integración	Se puede aplicar en todos los niveles.	Sistema.
Participantes	Desarrolladores.		Desarrolladores, *testers*, *stakeholders*, ingenieros de calidad.
Tipos de proyecto de *software* en los que se usan	No involucran a los usuarios finales.	Están centralizados en las acciones de los usuarios.	Donde es importante la experiencia del cliente y el mercado tiene gran competencia.
Herramientas utilizadas	*csUnit, NUnit, DocTest, PyUnit; JUnit, TestNG, PHPUnit, RSpec*.	*Cucumber, Jbehave, Instict, beanSpec* y *Behat*.	*JUnit, Selenium, Cucumber, Fitnesee, Concordion, robot framework*.

4. Conclusión

Como producto de la investigación realizada, se presentaron tres prácticas de desarrollo de *software* basados en pruebas: *Test-Driven Development, Behaviour-Driven Development* y *Acceptance Test Driven Development*. Se revisaron cada una de estas prácticas y se obtuvo que, al considerar las pruebas de *software* desde etapas iniciales, se guía el proceso de desarrollo, de manera tal, que las pruebas de *software* son un aspecto fundamental para cada característica del sistema. Ejemplo de ello, es el método TDD, donde se empieza con un producto mínimo de código que cumpla con el requisito y luego se pasa a la refactorización. Sin embargo, para evitar que al proceder con el método TDD existan inconsistencias con los requisitos establecidos y se fabrique un código que no satisfaga las necesidades de los *stakeholders*, se introdujo el conjunto de prácticas de ATDD y BDD. El primero, enfocado en mejorar la comunicación entre *testers* y desarrolladores pasando los requisitos a pruebas de aceptación. Mientras que el segundo, se centra en establecer una comunicación sólida con los *stakeholders* integrándolos en la etapa de definición de requerimientos y procediendo a crear escenarios en lenguaje natural para maximizar la comprensión del requerimiento de parte de todos los involucrados en el desarrollo. En relación con lo anterior, se comprobó que, al aplicar estas prácticas en el ciclo de vida del *software*, ya sea en metodologías ágiles o tradicionales, se obtienen resultados de mejor calidad, existe una mayor satisfacción por parte de los involucrados, al igual que, se logra incrementar la efectividad del *software* como tal. Finalmente, se recomienda analizar el comportamiento y adaptación de las practicas investigadas en referencia a proyectos de diferente extensión, debido a que puede de ser de gran valor conocer sus ventajas y desventajas al ser aplicadas en estas condiciones.

CONFLICTO DE INTERESES

Los autores declaran no tener algún conflicto de interés.

REFERENCIAS

[1] K. Beck, (1996, Mar.), “Extreme Programming”, [Online], Available: http://www.extremeprogramming.org/.

[2] M. C. Fontela, (2011, Jun.), “SEDICI”, [Online], Available: http://sedici.unlp.edu.ar/bitstream/handle/10915/4216/Documento_completo.pdf?sequence=1&isAllowed=y.

[3] O. Dieste, E. Fonseca, G. Raura y P. Rodríguez, (2015), “Revista Latinoamericana de Ingeniería de *Software*”, [Online], Available: http://revistas.unla.edu.ar/software/article/view/706.

[4] J. Anderson, P. Lawson, M. Renschler y M. Lange, (2002), “csUnit”, [Online], Available: http://www.csunit.org/.

[5] .NET Foundation, (2019), “NUnit”, [Online], Available: https://nunit.org/.

[6] Python, (2020, Mar. 26), “doctest — Test interactive Python examples”, [Online], Available: https://docs.python.org/3/library/doctest.html.

[7] K. Beck y E. Gamma, (2001, Ago. 10), “PyUnit - the standard unit testing framework for Python”, [Online], Available: http://pyunit.sourceforge.net/.

[8] K. Beck, E. Gamma, D. Saff y K. Vasudevan, (2021, May. 15), “Junit”, [Online], Available: https://junit.org/junit5/.

[9] C. Beust y T. Team, (2020, Sep. 19), “TestNG”, [Online], Available: https://testng.org/.

[10] S. Bergmann, (2021, Feb. 2), “Welcome to PHPUnit!”, [Online], Available: https://phpunit.de/.

[11] S. Baker, (2018, Ago. 4), “RSpec”, [Online], Available: https://rspec.info/.

[12] D. North, (2006), “Introducing BDD”, [Online], Available: http://dannorth.net/introducing-bdd.

[13] J. Smart, *BDD in Action: Behavior-driven development for the whole software lifecycle*, New York: Simon and Schuster, 2014.

[14] E. Evans, *Domain-Driven Design*, Boston: Addison-Wesley Professional, 2003.

[15] Jbehave, (2003), “Jbehave”, [Online], Available: https://jbehave.org/.

[16] Google | Code, “Instinct”, [Online], Available: https://code.google.com/archive/p/instinct/.

[17] SourceForge, (2021), “beanSpec”, [Online], Available: https://sourceforge.net/projects/beanspec/.

[18] Phpspec, “phpspec”, [Online], Available: http://www.phpspec.net/en/stable/.

[19] K. Kudryashov, (2016), “Behat”, [Online]. Available: https://docs.behat.org/en/latest/.

[20] “Cucumber- Behaviour-Driven Development”, Cucumber, [Online], Available: https://cucumber.io/docs/bdd/.

[21] Agile Alliance, “Acceptance Test Driven Development (ATDD)”, [Online], Available: https://www.agilealliance.org/glossary/atdd/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'acceptance*20test~'atdd))~searchTerm~'~sort~false~sortDirec.

[22] Global Logic, (2012, Mayo 7), “Acceptance TDD: una práctica clave en equipos ágiles”, [Online], Available: https://www.globallogic.com/latam/insights/blogs/acceptance-tdd-una-practica-clave-en-equipos-agiles/.

[23] D. H. Steinberg, *Extreme Software Engineering A Hands-On Approach,* Michigan: Prentice-Hall, Inc., 2003.

[24] R. Tejera, (2018, Nov. 2), “Información, Programación y Electrónica”, [Online], Available: https://rubentejera.com/desarrollo-dirigido-por-tests-de-aceptacion-atdd/.

[25] N. Koudelia, (2011, Dic. 6), "Acceptance Test-Driven Development", [Online], Available: https://jyx.jyu.fi/bitstream/handle/123456789/37392/URN%3aNBN%3afi%3ajyu-201202161200.pdf?sequence=1&isAllowed=y.
[26] *Software* Freedom, (2021), "Selenium", [Online], Available: https://www.selenium.dev.
[27] FitNesse. (2021). "FitNesse", [Online], Available sofware: http://fitnesse.org/.
[28] Concordion, (2020), "Concordion", [Online], Available: https://concordion.org/.
[29] Robot Framework Foundation, (2021), "Robot Framework", [Online]. Available: https://robotframework.org.