Test Driven Development

Fortalezas y Debilidades

Alejandro Araújo
(alar@bipbip.com.uy)

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Resumen

Test Driven Development (Desarrollo Dirigido por Pruebas) es una disciplina de diseño y programación, donde cada nueva línea de código que escribe un programador es en respuesta a una prueba que ha fallado, también escrita por el programador. Sin ser definida como una metodología de pruebas se apoya fuertemente en las pruebas unitarias y también en pruebas de aceptación, basándose en prácticas formalizadas por *Extreme Programming.* Ha despertado gran interés tanto en ámbitos académicos como de la industria, encontrándose en la literatura investigaciones sobre su aplicación, aunque aún en limitado número, ya sea mediante experimentos controlados o reportes acerca de su práctica en la industria. En el presente artículo se abordará *Test Driven Development* en general, el dominio de su aplicación y se analizarán ventajas y desventajas de su adopción.

Palabras claves: Test Driven Development, Extreme Programming, Xunit, FIT.

Breve reseña histórica

Existen antecedentes de utilización esporádica de técnicas de tipo *Test Driven Development* (TDD) que datan de varias décadas atrás, citándose en particular las técnicas aplicadas en el proyecto *Mercury*[1] de la NASA en el comienzo de la década de 1960, que incluían la planificación y escritura de las pruebas previo a cada micro incremento [LB03], pero será a partir de la metodología *Extreme Programming* [BA04] [Bec199] [Bec299] que emerge otra vez, difundiéndose a nivel mundial.

Durante la década de 1980 y a comienzos de los 90, Kent Beck y Ward Cunningham colaboraron y refinaron sus prácticas con el propósito de lograr que el desarrollo de programas fuera más adaptativo y orientado hacia las personas [Fow05], apoyándose en ideas y prácticas existentes, con notorias influencias de las ideas de Chistopher Alexander[2], Takeuchi & Nonaka[3], Jacobsen[4] y otros, las cuales se constituyen en sus raíces [BA04 sec.2] [Bec299] En Marzo de 1996 Kent Beck fue contratado para revisar el proyecto C3 [BA04 cáp.17] [Fow07] en Daimler Chrysler, el cual fue restaurado bajo su liderazgo. Esto dio lugar a la formalización de la metodología *Extreme Programming,* colaborando estrechamente en dicha formalización Ward Cunningham y Ron Jeffries.

En el año 1999 Kent Beck publica el libro que puede considerarse como el manifiesto de la metodología: "*Extreme Programming Explained: Embrace Change*" [Bec199]. Una de las prácticas fundamentales de dicha metodología, "*Test First Programming*", se sustenta en TDD. En tanto, en el mismo año, Martin Fowler et all publican "*Refactoring, Improving the Design of Existing Code*", [Fow99] considerado una fundamental introducción a la técnica de reconstrucción de código, práctica que se aplica en TDD.

En Febrero de 2001 se marca otro hito, en Salt Lake City, Utah, al emitirse el "Manifiesto para Desarrollo Ágil de Software" [MA01] por parte de un grupo de metodologístas allí reunidos, entre los cuales estaban los proponentes de XP. *Test Driven Development* tendrá un gran impulso al ser adoptado por practicantes y defensores de dichas metodologías, con las cuales comparte entre otras características, el desarrollo iterativo e incremental en muy breves ciclos y el diseño simple inicial.

En los años 2002 y 2003, se publican libros que tratan directamente sobre TDD, presentando y formalizando la disciplina: "*Test Driven Development by Example*"[5] por Kent Beck [Bec02] y "*Test-driven development: A Practical Guide*"[6] por David Astels [Ast03].

Características

Test Driven Development (Desarrollo Dirigido por Pruebas) es una disciplina iterativa e incremental de diseño y programación, donde cada nueva línea de código se escribe en respuesta a una prueba fallida que el propio programador ha programado. Es parte

[1] http://www-pao.ksc.nasa.gov/history/mercury/mercury-overview.htm Este proyecto recogió la experiencia del exitoso proyecto sobre el jet x15 en 1950s y otros, como ser un proyecto para simulación celular en Motorola, 1958.
[2] "*The Timeless Way of Building*", Oxford University Press, 1979
[3] "*The New Product Development Game*". Harvard Business Review. 1986.
De notoria influencia en otra metodología agil:"Scrum" http://www.controlchaos.com
[4] "*Object-Oriented Software Engineering*" Addison Wesley. 1994
[5] Es el primer libro sobre el tópico, conteniendo un ejemplo básico, un ejemplo de Xunit y patrones para TDD.
[6] Guía practica con problemas y soluciones reales.

medular del desarrollo ágil de código derivado de XP, donde resulta en una práctica crítica, y de los principios del Manifiesto Ágil. Apoyada en la popularidad de XP, *Test Driven Development* ha tenido gran difusión mundial en los últimos años, pudiéndose adoptarse dentro de cualquier otro tipo de proceso de desarrollo de *Software*. Su principal objetivo es obtener "Código limpio que trabaje"[7] [Bec02.]

Siguiendo la categorización citada en [Mug03], TDD puede ser aplicado a dos niveles, a saber:

- Nivel de micro-iteraciones: En este nivel el desarrollo es guiado por pruebas unitarias escritas por el propio programador de la aplicación, donde los pasos son, según [Bec02]:
    - Rápidamente agregar una prueba.
    - Correr todas las pruebas y comprobar que solo la nueva falla.
    - Hacer un pequeño cambio.
    - Correr todas las pruebas y comprobar que pasan satisfactoriamente.
    - Reconstruir para remover duplicaciones.

  Lo anteriormente expuesto se puede resumir en el diagrama de actividad de la figura 1.

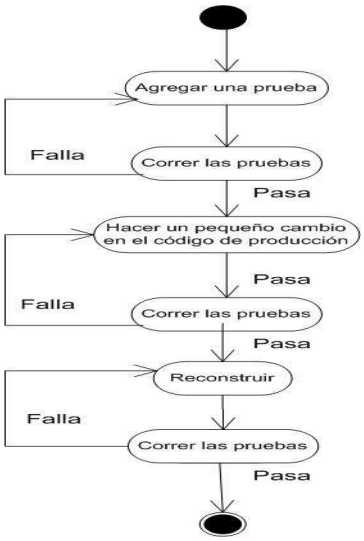

Figura 1. Pasos en un ciclo TDD

Las principales tareas se pueden agrupar en "Escribir y ejecutar pruebas", "Escribir código de producción" y "Reconstrucción", detallándose a continuación:

Escribir y ejecutar pruebas:

La escritura de las pruebas sigue varios patrones identificados y detallados por Beck en [Bec02]. En particular se destaca que las pruebas:

- Son escritas por el propio desarrollador en el mismo lenguaje de programación que la aplicación, y su ejecución se automatiza. Esto último es primordial para que obtenga un retorno inmediato, indicándose que el ritmo de cambio entre prueba-código-prueba esta pautado por intervalos de no más de diez minutos. La automatización también permite

[7] Frase acuñada por Ron Jeffries.

aplicar, en todo momento, la batería de pruebas, implicando pruebas de regresión inmediatas y continuas.
- Las pruebas se deben escribir pensando en primer lugar en probar las operaciones que se deben implementar.
- Deben ser aisladas, de forma que puedan correrse independientemente de otras, no importando el orden Este aislamiento determinará que las soluciones de código cohesivas y bajo acoplamiento, con componentes ortogonales[8].
- Deben de escribirse antes que el código que se desea probar.

Escritura de código:

La escritura de código es el proceso por el cual se logra hacer que la prueba unitaria pase [Sis06]. La propuesta de Beck contempla cuatro estrategias para salir rápidamente de la situación de falla, descritas como patrones en [Bec02]: Imitarlo (*Fake It*): Retornar un valor esperado, Triangulación, Implementación obvia y Uno a muchos.

En cuanto a la programación en si, la recomendación, extraída de la aplicación de la práctica en XP, consiste en programar por intención (*programming by intention*) lo cual implica no detenerse a pensar en como hacer una cosa, sino en lo que se debe hacer [Jef00 cáp.14].

Reconstrucción (*Refactoring*):

En Fowler et all [Fow99] se define la reconstrucción como una practica que consiste en mejorar el código existente, de manera disciplinada, sin alterar su comportamiento externo, para hacerlo más fácil de entender y modificar. La mejora de código se logra mediante la aplicación de una serie de consejos entre los cuales de destaca la eliminación de duplicación, la división en segmentos menores, la estructuración de manera que resulte más conveniente y la aplicación de las ventajas que brinde el lenguaje de programación[9]. La reconstrucción cumple un rol sustancial como complemento del diseño.

El orden de ejecución de las tareas se resume con la frase Rojo/Verde/Reconstrucción (Red/Green/Refactor). En Rojo se está en la etapa en que una prueba falló, en Verde en la etapa cuando la aplicación pasa las pruebas.

A modo de resumen, se establece que TDD a este nivel sigue tres grandes leyes [Mar07]:
- No se escribe código de producción salvo que se hubiera escrito previamente una prueba unitaria que falló.
- No se escribe más de una prueba unitaria que sea suficiente para fallar.
- No se escribe más código de producción que el suficiente para pasar la prueba unitaria

[8] En el sentido de independencia entre objetos. En el libro "*The Pragmmatic Programmer: From Journeyman to Master*" se expone definición y discusión de la importancia de dicho tópico [HT99 pp.43-45].
[9] En particular referido a facilidades de la programación Orientada a Objetos (p/ej: Polimorfismo y Herencia).

- Nivel Iteración o Funcional: El desarrollo es guiado por pruebas de aceptación. Este nivel, mencionado en [Bec02] como ATDD (*Aceptance TDD*), consta de los siguientes pasos:
    - El usuario especifica pruebas antes que las funcionalidades sean implementadas.
    - Una vez que el código es escrito la prueba sirve como un criterio de aceptación.

  La construcción de las pruebas es también a este nivel un proceso evolutivo, salvo que el retorno se da en ventanas de tiempo más extendidas, a nivel de iteración, y están fundamentalmente en manos del usuario apoyado por verificadores y programadores. Dado que se requiere la automatización de las mismas se lo conoce también como "*Executable Aceptance TDD*" (EATDD) o "*StoryTest-Driven Development*".

En *Test Driven Development* las pruebas siempre actúan, en ambos niveles, como estímulos para el desarrollo; el cual responde para satisfacerlas. Tal como se expresa en [Bec02] el punto de vista de las pruebas por parte de TDD es pragmático, son estas el medio utilizado para obtener certezas, vencer los temores y guiar el desarrollo.

Por último se desea resaltar que si bien *Test Driven Development* se ha definido como una disciplina de análisis, diseño y programación, no como una disciplina de pruebas:
"una de las ironías de TDD es justamente no ser una técnica de pruebas" [Bec02]
y sus proponentes hacen especial hincapié en destacar este aspecto, sus prácticas centrales pueden ser analizadas desde el punto de vista de la prueba de programas, haciéndose mención a ella además como "*Extreme Testing*" [Mye04] abarcando las pruebas unitarias y de aceptación.

Herramientas[10]

Para utilizar la disciplina se debe contar con herramientas que permitan la automatización de las pruebas. Beck advierte, en [Bec199] que "una porción de código sin un programa que la pruebe automáticamente simplemente no existe". A tales efectos se han desarrollado marcos para definir y ejecutar las pruebas, tanto unitarias como de aceptación, ya que la automatización se considera imprescindible. Es lo que permite la ejecución rápida y eficaz de todas las pruebas existentes con inmediato retorno al programador, a los efectos de brindarle las necesarias certezas y posibilitar la prueba de regresión, evitando además el ciclo perverso de más presión, menos pruebas, más fallas, provocado por el apremio en los plazos [Bec199].

Las herramientas disponibles se especializan en general para pruebas unitarias o pruebas de aceptación, aunque la división no es estricta y podrían utilizarse en ambos niveles, ya sea mediante la incorporación de extensiones y bajo ciertas restricciones. En particular se han desarrollado varios marcos para pruebas *open-source*, con participación en su gestación de los proponentes principales del emergente TDD.

Xunit: Familia de marcos para pruebas unitarias

[10] Las herrmientas han sido desarrolladas utilizando TDD.

Beck escribió un marco de pruebas para SmallTalk llamado "SUnit"[11] [Sun07], que fue extendido luego por J. Pelrine y otros durante varias iteraciones. Dicho marco se expone originalmente en el trabajo titulado "*Simple SmallTalk Testing: With Patterns*" [Bec98]. Luego de esto, Beck trabajando junto a Erich Gamma, portan el marco a Java [Jav07] dando lugar a "JUnit" [Jun07]. A partir de allí comienza a ganar aceptación y amplia difusión, habiendo sido portado a múltiples plataformas y lenguajes de programación, constituyendo la familia de marcos para pruebas unitarias conocida como Xunit[12] [Jun07].

En la figura 2 se muestra a modo de ejemplo como luce JUnit durante una prueba.

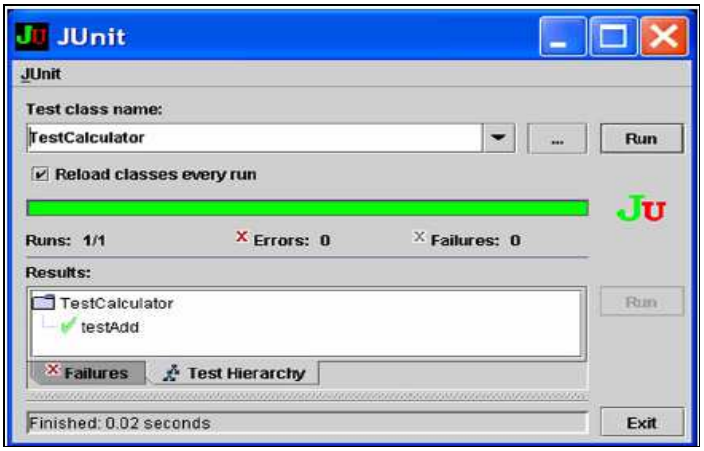

Figura 2. JUnit TestCalculator [MH04].

El programador utiliza el marco extendiendo o importando sus clases, utilizando ciertos 'marcadores' (*tags*) para indicar a la herramienta los métodos de prueba, que se escriben en el propio lenguaje de la aplicación. Mediante sentencias apropiadas tales como "*Assert*", el programador puede validar los datos obtenidos contra los resultados esperados y notificar de la condición encontrada. Al ejecutarse bajo el marco este busca utilizando "*Reflection*"[13] dentro de los ejecutables las clases, variables y el resto de la información que necesita para ejecutar las pruebas, notificando al programador del resultado de las mismas con un símil del semáforo: Rojo no pasó la prueba, Verde pasó satisfactoriamente, Amarillo dio alguna excepción. El desarrollador puede agrupar las pruebas en conjuntos (*Suites*) para organizar su ejecución en una corrida.

Existe una gran cantidad de extensiones e IDE's a los que puede integrarse. El sitio http://www.junit.org [Jun07] está dedicado a los desarrolladores que utilizan JUnit o algunos de los otros marcos de la familia XUnit.

FIT: Marco para pruebas de aceptación

FIT [FIT107] es una herramienta creada por Ward Cunningham con el objetivo de automatizar las pruebas de aceptación y mejorar tanto la comunicación como la

[11] http://sunit.sourceforge.net/
[12] La X se sustituye generalmente por la primer letra, o primera y segunda letra,del nombre del lenguaje o plataforma a la que ha sido portado.
[13] Proceso de inspección y modificación de un programa en tiempo de ejecución. Concepto desarrollado por Smith,B.C. en: "*Procedural Reflection in Programming Languages*", PhD Thesis, Massachusetts Institute of Technology, 1982. http://hdl.handle.net/1721.1/15961

colaboración en el desarrollo de *software*. Según la descripción expresada en el libro "*Fit for Developing Software: Framework for Integrated Tests*" [MC05 p.1]:

"*FIT (Framework for Integrated Tests) es un poderoso marco de trabajo para pruebas automatizadas…FIT es especialmente indicado para probar desde una perspectiva de negocios, usando tablas para representar las pruebas y para reportar los resultados de la comprobación automática de dichas pruebas. Este formato tabular permite que personas sin conocimientos de programación escriban casos de prueba y guíen el desarrollo general del sistema necesario. FIT es un marco de trabajo de propósito general abierto y fácil de extender para expresar diferentes ordenes de pruebas."*

Es una herramienta de pruebas dirigidas por palabras claves y datos, que implementa los siguientes pasos:

- Lee tablas que contienen información para la prueba y los resultados esperados (ejemplos).
- Interpreta cada tabla con una clase (conocida como "*Fixture*").
- Compara el resultado de correr el programa bajo prueba con los ejemplos suministrados en las tablas.

Las "*fixtures*" son escritas por los programadores, en el mismo lenguaje que el código de producción. Son muy pequeñas porciones de código que se encargan de actuar como el pegamento entre las "*fixtures*" suministradas por el marco y las tablas con la especificación de las pruebas entradas por el usuario. En general extienden las "*fixtures*" del marco o ejecutan bajo ellas. Al igual que los marcos Xunit busca utilizando "*Reflection*" dentro de los ejecutables las clases, variables y el resto de la información que necesita para ejecutar las pruebas, notificando al programador del resultado con similar código de colores.. El desarrollador puede agrupar las pruebas en conjuntos ("*Suites"*) para organizar su ejecución en una corrida. Presenta la particularidad que no se detiene la ejecución de las pruebas ante un error.

Dado que FIT no provee una interfase gráfica para su utilización se han elaborado extensiones para cargarlo en IDE's, tales como Eclipse [Ecl07], o se han desarrollado productos para mantener y organizar los casos de pruebas, de los cuales notoriamente el más conocido es FitNesse [MMW06].

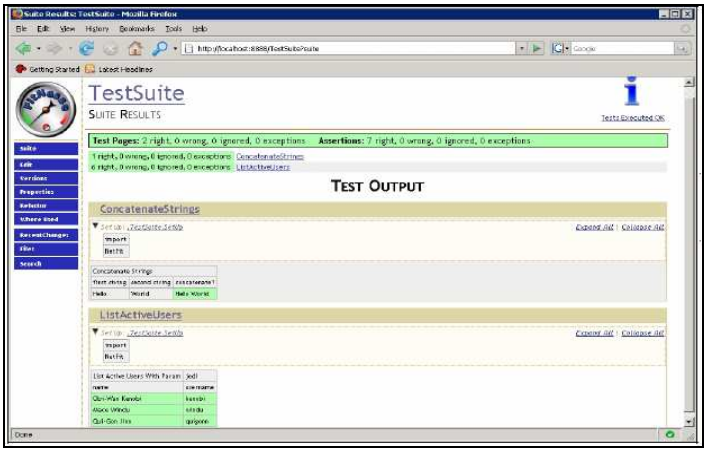

Figura 3. FitNesse[Adz107].

El sitio http://c2.fit.org [Cun207] esta dedicado a los desarrolladores, usuarios y verificadores que utilizan FIT y http://www.fitnesse.org [MMW06] a aquellos que utilizan FitNesse.

Fortalezas y Debilidades

Test Driven Development presenta , como toda disciplina, defensores y detractores, fortalezas y debilidades; En la literatura es posible encontrar mucha información acerca de estas, especialmente desde un punto de vista teórico. Sería muy útil contar con abundante evidencia empírica documentada, para obtener criterios objetivos de valorización y para medir adecuadamente los beneficios y perjuicios de su utilización así como las mejores prácticas para determinar cuando resulta oportuno adoptarla y para que tipo de proyectos. Sin embargo, solo recientemente la información de este tipo se está exponiendo y sumarizando en publicaciones, como ser [Fit207],[JM07],[MM07] [REA05],[Sin06]] aunque probablemente sea un importante indicio de que se contará con dicha información en el mediano plazo, partiendo de la base que el resurgimiento y formalización de la disciplina data de hace pocos años. Dada esta situación se opinará acerca de algunas de las fortalezas y debilidades desde un punto de vista teórico.

Fortalezas

- **Al elaborar las pruebas primero** el programador no siente que la prueba destruye su código. Se invierte la posición, ahora es el código el que salva la prueba, convirtiéndose el acto destructivo de la prueba en una carrera de salvar obstáculos, paso a paso. Por otra parte, y tal como destacaba Myers en 1975 [Mey04], siendo las pruebas un proceso destructivo se establece una dificultad sicológica desde el inicio para su correcta elaboración por parte de los programadores; hecho que TDD ayuda a resolver.

- **Al elaborar sus propias pruebas** no somete su código a pruebas ajenas. Nuevamente se pone la carga en la responsabilidad de elaborar buenos obstáculos para un código saludable que deberá sortearlos, sin la presión de la competencia.

- **Al automatizar** el programador rápidamente obtiene retorno acerca del estado de salud de la aplicación. Si introduce mejoras en el código o nuevas funcionalidades sabrá rápidamente si pasan todos las pruebas. Esto reduce el stress y el temor. La automatización así lograda está escrita en el mismo lenguaje en que programa, y acompaña al código. No necesita aprender lenguajes nuevos.

- **En cuanto al método,** siguiendo el razonamiento empleado por Mugridge en [Mug03], este puede ser explorado desde el valor del método científico como una metáfora para *Test Driven Development*, especialmente al considerar la evolución de la teoría y el rol de la reproducibilidad en la experimentación, lo que le da un enorme valor a la disciplina. En las siguientes figuras se representa dicha metáfora alineando las prácticas con el método científico:

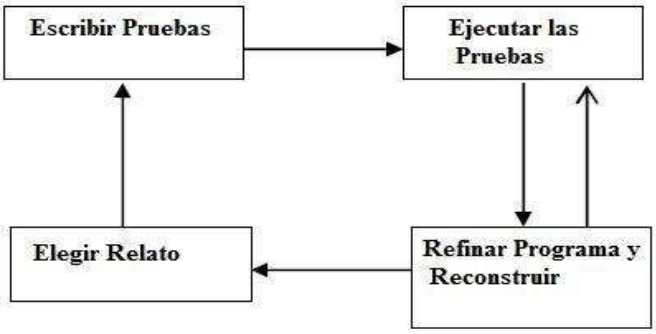

Figura 4. TDD [Mug03].

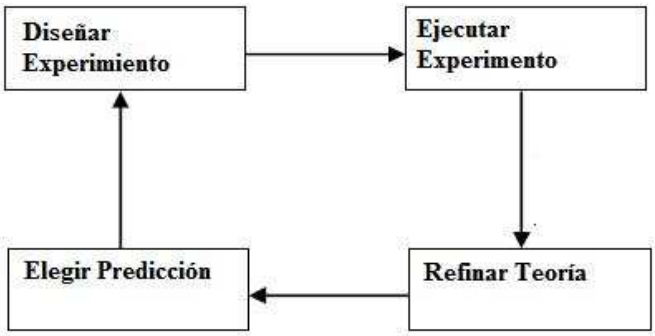

Figura 5 Método científico [Mug03].

Sin llegar a sostener, al contrario de Mugridge [Mug03], que el resto de los métodos están equivocados, es razonable coincidir en la importancia de esta correspondencia.

- **Al prevenir defectos**, mediante el diseño continuo de pruebas, extendiendo la aseveración de Beizer acerca que el acto de diseñar pruebas es de las técnicas más efectivas de prevención de defectos [Bei90].

Debilidades

- **Los ciclos extremadamente cortos y el cambio entre pruebas y codificación** de producción hacen que muchos programadores lo sientan como contrario a su intuición y desgastante. Exige además disciplina, responsabilidad y coraje.

- **La imposibilidad de adopción** cuando se utilizan lenguajes de programación que no cuentan con el marco de pruebas unitarias, o cuando se utilizan herramientas de alto nivel (CASE) para la generación de código y asistencia al desarrollo que no lo integran. Si bien esto puede resolverse en muchos casos con desarrollo propio, no siempre es posible por problemas de costo y tiempo.

- **Las dificultades o imposibilidad** para su adopción cuando se utilizan herramientas de alto nivel (CASE) para la asistencia al desarrollo o lenguajes de programación que no adhieren al paradigma de la orientación a objetos.

- **En las pruebas de aceptación**, donde es más dificultosa la automatización, especialmente pues la prueba puede ser asimilada a un ciclo funcional, lo que acrecienta la complejidad de los casos, con altamente probable intervención de interfases, especialmente gráficas (GUI), desde las cuales la aplicación interactúa con el usuario. También es de particular importancia a este nivel el tiempo que transcurre entre la elaboración de la prueba y su ejecución, que ya no puede medirse en minutos, sino en días o iteraciones, lo cual quita ritmo al proceso y disminuye los aspectos positivos del rápido retorno.

Conclusiones

Seguramente TDD no se convertirá en la bala de plata[14] [Bro95 cáp.16], pero sus fortalezas parecen aventajar en mucho a sus debilidades, la mayoría de las cuales radicarían en los problemas que presenta la automatización de las pruebas. De allí la importancia de investigar, proponer y construir mecanismos que permitan su utilización en ambientes de desarrollo que usan modernas herramientas de programación bajo las cuales no es posible aún su utilización. La baja penetración en Uruguay de esta disciplina es probable que se deba, entre otras razones, a la falta del marco adecuado relacionado a las herramientas de desarrollo más utilizadas.

Bibliografía

[Adz107] Adzic, G. “Getting Fit with Net. Quick Introduction to Testing .Net Applications with FitNesse”, Version 0.2, http://www.gojko.net/FitNesse/fitnesse.pdf 02/2007.

[Ast03] Astels, D. “Test-Driven Development: A Practical Guide“,ISBN 0131016490, Prentice Hall, 2003.

[BA04] Beck, K. , Andres C. ““Extreme Programming Explained, Embrace Change 2nd Edition”, ISBN 0-321-27865-8, Addison Wesley, 2004.

[Bec98] Beck, K. “Simple Smalltalk Testing: With Patterns”, First Class Software, Inc., http://www.xprogramming.com/testfram.htm, 1998.

[Bec02] Beck, K. “Test Driven Development by Example”, ISBN 032-114653-0, Addison Wesley, 2002.

[Bec199] Beck, K. “Extreme Programming Explained, Embrace Change”, ISBN 201-61641-6, Addison Wesley, 1999.

[Bec299] Beck, K. “Embracing Change with Extreme Programming”, IEEE Computer, Octubre 1999, pp. 70-77.

[Bei90] Beizer, B. “Software testing techniques”, 2nd. Edition, ISBN 0-442-20672-0, Van Nostrand Reinhold Co, 1990.

[Bro95] Brooks, F. P. “The Mythical Man-Month: Essays on Software Engineering”, Anniversary Edition, ISBN 0-201-83595-9, Addison Wesley, 1995.

[Cun207] Cunningham W. “Welcome Visitors”,”Framework for Integrated Tests”, http://fit.c2.com/ 28/01/2007.

[Ecl07] Eclipse Foundation “Eclipse - an open development platform”, http://www.eclipse.org, 2007.

[14] *“No Silver Bullet-Essence and Accident in Software Engineering”* Brooks F.P.

[FIT107] Cunningham W. “Framework for Integrated Test”, http://sourceforge.net/projects/fit/ , 2007.

[FIT207] Deng, C., Wilson P. , Maurer F. “FitClipse: A Fit-based Eclipse Plug-in For Executable Acceptance Test Driven Development”, Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007), Come, Italy,http://ebe.cpsc.ucalgary.ca/ebe/attach/Publications.2007/Deng_Wilson_Maurer_FitClipse.pdf ,2007.

[Fow05] Fowler M. “The New Methodology”, http://www.martinfowler.com/articles/newMethodology.html#xp, 12/2005.

[Fow07] Fowler M. “C3”, http://www.martinfowler.com/bliki/C3.html, 06/06/2007.

[Fow99] Fowler M. et all “Refactoring: Improving the Design of Existing Code”, 1st edition, ISBN 0201485672, Addison-Wesley Professional, 1999.

[HT99] Hunt, A. , Thomas D. “The Pragmatic Programmer: From Journeyman to Master”, 1st edition, ISBN 0-201-61622-X, Addison Wesley, 1999.

[Jav07] Sun Microsystems Java, “Java Technology. The Power of Java” http://www.sun.com/java/, 05/2007.

[Jef00] Jeffries, R. , Anderson A. , Hendrickson, C. , Jeffries, R. E., “Extreme Programming Installed”, 1st edition, ISBN 0-201-70842-6 Addison Wesley, 2000.

[JM07] Jeffries, R , Melnik, G. “Guest Editors' Introduction: TDD--The Art of Fearless Programming “ ,IEEE Computer, May/Jun 2007, Vol.24, Issue 3, pp. 24-30.

[Jun07] Junit.Org,, Object Mentor, http://www.junit.org/index.htm , 07/2007.

[LB03] Larman, C. , Basili, V. “Iterative and Incremental Development: A Brief History”, IEEE Computer; Volume 36, Issue 6, pp 47 – 56, Junio 2003.

[MA01] Manifiesto for Agile Software Development, http://www.agilemanifesto.org, 2001.

[Mar07] Martin, R.C. “Professionalism and Test-Driven Development”, IEEE Computer; Volume 24, Issue 3, pp 32 - 36, May-Jun 2007

[MC05] Mugridge R.,Cunningham W. ”Fit for Developing Software: Framework for Integrated Tests”, ISBN 0-321-26934-9, Prentice Hall PTR, 2005.

[MH04] Massol, V. , Husted, T. “Junit in Action”, ISBN 1-930110-99-5, Manning Publications Co.,2004.

[MM07] Melnik, G. , Maurer, F. “Multiple Perspectives on Executable Acceptance Test-Driven Development”, Department of Computer Science, University of Calgary, Canada, http://ebe.cpsc.ucalgary.ca/ebe/attach/Publications.2007/XP2007_Melnik_Maurer.pdf, 2007

[MMW06] Martin, R., Martin, M., Wilson, P. "Welcome to FitNesse", http://www.fitnesse.org, 01/12/2006.

[Mug03] Mugridge, R. “Test Driven Development and the Scientific Method”, Department of Computer Science,University of Auckland,New Zealand, http://agile.openmex.com/agile2007/2003/files/P6Paper.pdf, 03/2006.

[Mye04] Myers G. “The Artof Sotware testing, 2nd edition”, ISBN 0-471-46912-2, John Wiley & Sons Inc., 2004.

[Rea05] Read, K. “Supporting Agile Teams of Teams via Test Driven Design”, Master Thesis, Departament of Computer Science, University of Calgary, Calgary, Canada, http://ebe.cpsc.ucalgary.ca/ebe/attach/Publications.2005/Read2005.pdf 02/2005

[Sin06] Siniaalto, M. “Test driven development: empirical body of evidence”, “Agile Software Development of Embedded Systems”, Versión 1.0, ITEA, Agile VTT, www.agile-itea.org/public/deliverables/ITEA-AGILE-D2.7_v1.0.pdf, 3/03/2006.

[Sun07] Camp SmallTalk SUnit Project “Sunit. The mother of all unit testing framework”, http://sunit.sourceforge.net, 08/2007.