UNIVERSIDAD PRIVADA DE TACNA


INGENIERIA DE SISTEMAS

TEMA:

# Test-Driven Development

## CURSO:

BASE DE DATOS II

## DOCENTE(ING):

Patrick Jose Cuadros Quiroga

Integrantes:

| | |
|---|---|
| Marko Antonio RIVAS RIOS | (2016055461) |
| Jorge Luis MAMANI MAQUERA | (2016055236) |
| Andree Ludwed VELASCO SUCAPUCA | (2016055286) |
| Yofer Nain CATARI CABRERA | (2017059289) |
| Adnner Sleyder ESPERILLA RUIZ | (2015050543) |
| Jesus ESCALANTE ALANOCA | (2015050641) |

# Índice

# 1. Objetivos del Desarrollo Orientado A Pruebas (TDD)

El propósito de esta técnica tiene los siguientes 3 objetivos básicos:

**1. Minimizar el número de bugs:** Mientras más bugs salgan menos rentable es el proyecto, porque corregir los bugs es tiempo que se puede invertir mejor en otras tareas. Si no hay bugs se puede conseguir una mayor rentabilidad de la aplicación.

**2. Implementar las funcionalidades justas que el cliente necesita:**

Es muy común que cuando se explican los requisitos de una aplicación o las especificaciones en la fase de análisis y diseño se esboza una aplicación, y el diseñador por su experiencia con funcionalidades pensando que van a ser útiles para la aplicación, para el cliente o para otros componentes. Sin embargo casi el 95 % de esas funcionalidades extras no se usan en ninguna parte de la aplicación, eso implica tiempo invertido desarrollando algo que no ha llegado a nada. El objetivo de TDD es eliminar ese código innecesario y esas funcionalidades que no ha pedido el cliente, con lo cual redunda en eficiencia, tanto en el trabajo y como en la rentabilidad de la aplicación.

**3. Producir software modular, altamente reutilizable y preparado para el cambio:** Esto es realmente más técnica, porque con buenos hábitos de programación siempre se logra que el proyecto sea modular y reutilizable. Prepararlo para el cambio es una característica que no se consigue siempre y que con TDD si, ya que muchas veces cuando se tiene que cambiar la funcionalidad de la aplicación se tiene que refactorizar código ajeno, trabajar con código complicado, entre otras cosas; en cambio con TDD se tiene la confianza de que cuando se haga cambios no se van a estropear las funcionalidades que ya se tienen. Esto se consigue ya que la forma funcional de TDD es que primero se construye la prueba y luego el código hace que todo lo que surja en él ya este testeado, así que cualquier cambio que se vaya a introducir estará cubierto por los tests y si llegas a dañar algo alguno de ellos reaccionara cuando se ejecuten.

Todo esto cambia un poco la mentalidad tradicional que es: primero analizar los requisitos, luego hacer un diseño completo y profesional, después empezar a codificar y por último testear. Lo que hay que hacer es que, en vez de planear tareas pensar en ejemplos y datos concretos, ya que en eso se basa los test, tener parámetros de entrada y de salida y luego ver si la respuesta es lo que se esperaba. Si con TDD consigues tener en las especificaciones o requisitos una lista de ejemplos muy completa, que sean concretos, que elimine cualquier tipo de ambigüedad y que se puedan transformar en pruebas, al final tendrás una batería de tests que te cubrirán todas las funcionalidades y el código resultante estará 100 % cubierto por dichos test.

## 2. ¿Qué es Desarrollo Orientado A Pruebas (TDD)?

Esta técnica llamada TDD (Test Driven Development), se puede definir como un proceso de desarrollo de software que se basa en la idea de desarrollar unas pequeñas pruebas, codificarlas y luego refactorizar el código que hemos implementado anteriormente. Podemos decir que esta técnica e implementación de software está dentro de la metodología XP donde deberíamos de echarle un ojo a todas sus técnicas, tras leer varios artículos en un coincido con Peter Provost con un diseño dirigido o implementado a base de ejemplos hubiese sido mejor pero TDD se centra en 3 objetivos claros:

- Una implementación de las funciones justas que el cliente necesita y no más, solamente las funciones que necesitamos, estoy cansado de duplicar dichas funciones para que hagan lo mismo

  Mínimos defectos en fase de producción

- Producción de software modular y sobre todo reutilizable y preparado para el cambio

  Esta técnica se basa en la idea de realizar unas pruebas unitarias para un código que nosotros debemos construir, Nuestro TDD lo que nos dice es que primero los programadores debemos realizar una prueba y a continuación empezar a desarrollar el código que la resuelve. El método que debemos seguir a para empezar a utilizar TDD es sencillo, Nos sirve para elegir uno de los requisitos a implementar, buscar un primer ejemplo sencillo, crear una prueba, ejecutarla e implementar el código mínimo para superar dicha prueba. Obviamente la gracia de ejecutar la prueba después de crearla es ver que esta falla y que será necesario hacer algo en el código para que esta pase. El ciclo de desarrollo de TDD es empezar la prueba, en test realizar un test, revisar el código y pasar el refactor.

  **Crear la prueba o test**

- Ejecutar los tests: falla (ROJO)
- Crear código específico para resolver el test
- Ejecutar de nuevo los tests: pasa (VERDE)
- Refactorizar el código
- Ejecutar los tests: pasa (VERDE)

  **Personalmente, añadiría lo siguiente:**

- Incrementa la productividad.
- Nos hace descubrir y afrontar más casos de uso en tiempo de diseño.
- La jornada se hace mucho más amena.
- Uno se marcha a casa con la reconfortante sensación de que el trabajo está bien hecho.

  Ahora bien, como cualquier técnica, no es una varita mágica y no dará el mismo resultado a un experto arquitecto de software que a un programador junior que está empezando. Sin

embargo, es útil para ambos y para todo el rango de integrantes del equipo que hay entre uno y otro. Es una técnica a tener en cuenta en el desarrollo web y sobre todo en el desarrollo de ingeniería software donde debemos tener en cuenta muchos fallos antes de pasar a producción.

# 3. Sub-prácticas, Pruebas Unitarias y Simulación de objetos y Tipos de Test

### 1. Sub-prácticas

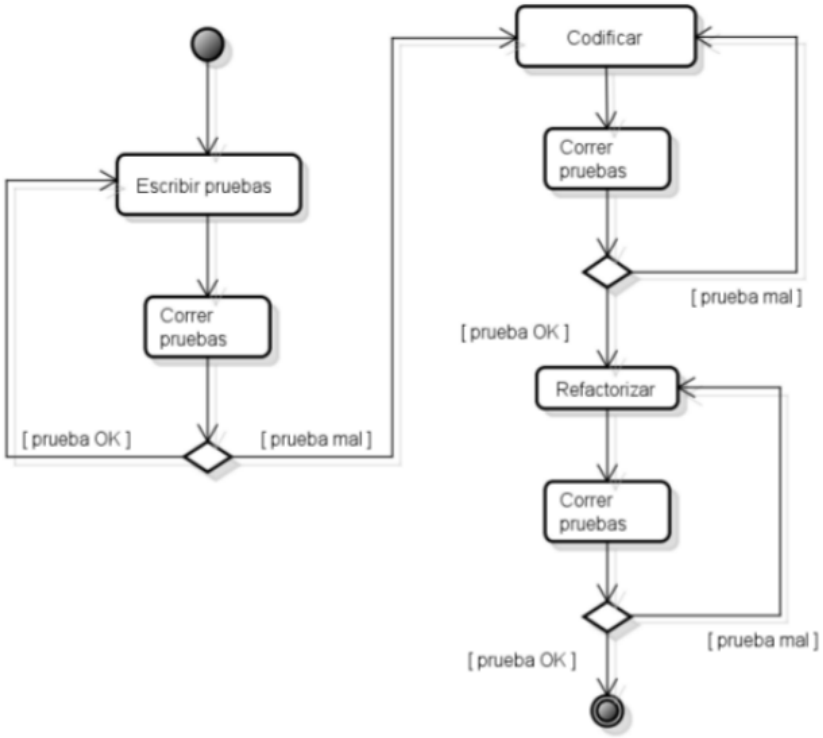
**- Test-First:** Las pruebas se escriben antes de escribir el propio código, y las mismas son escritas por los propios desarrolladores, esto busca que los mismos logren un entendimiento de lo que deben desarrollar mediante la construcción del código que lo va a probar.

**- Automatización:** Las pruebas deben ser escritas en código, y esto permite que se ejecuten automáticamente las veces que sea necesario, y el solo hecho de ejecutar las pruebas debe mostrar si la ejecución fue correcta o no.

**- Refactorizar el código:** Permite mantener la calidad de la arquitectura, se cambia el diseño sin cambiar la funcionalidad, manteniendo las pruebas como reaseguro.

### 2. Pruebas Unitarias y Simulación de objetos

Otra alternativa para realizar la simulación, consiste en conseguir probar el código unitariamente, esto significa aislarse de todos los recursos externos, es decir no depender de la infraestructura de red, o de un determinado entorno, o incluso del proceso que ejecutará nuestro código cuando esté en producción. Cargaremos las pruebas y el código a probar en un proceso encargado de gestionar la ejecución de las pruebas.



### 3. Tipos de Test

**- Test de aceptación:** es un test que permite comprobar que se está cumpliendo con un requerimiento del negocio. Son pruebas escritas en lenguaje del cliente pero que puede ser ejecutado con la máquina. Esto nos permite probar que el software que estamos desarrollando cumple con las expectativas del cliente y de los usuarios.

**- Test funcionales:** Si bien, siendo estrictos, todos los tests son funcionales ya que prueban alguna funcionalidad, esta expresión es utilizada para determinar a aquellas pruebas que agrupan a varios tests de aceptación y prueban alguna funcionalidad del negocio propiamente dicha.

**- Test de sistema:** Integra varias partes del sistema, incluso puede probar toda la aplicación o varias funcionalidades juntas. Estos tests se comportan de manera similar y buscan emular el comportamiento de los usuarios del sistema.

**- Test unitarios:** Son los tests ineludibles, son los necesarios y los más importantes para los desarrolladores, todo test unitario debe ser rápido, atómico, inocuo e independiente, sino cumple con estas cuatro premisas no es un test unitario.

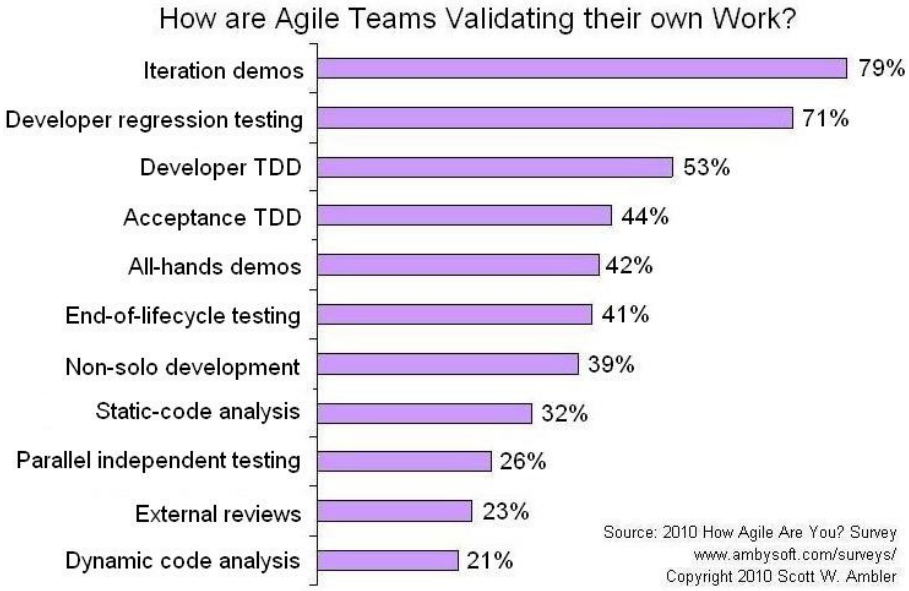
# 4. Proceso de diseño de software combinado con TD

- 1.El Cliente escribe su historia de usuario.
- 2.Se escriben junto con el cliente los criterios de aceptación de esta historia, desglosándolos mucho para simplificarlos todo lo posible.
- 3.Se escoge el criterio de aceptación más simple y se traduce en una prueba unitaria.
- 4.Se comprueba que esta prueba falla.
- 5.Se escribe el código que hace pasar la prueba.
- 6.Se ejecutan todas las pruebas automatizadas.
- 8.Se refactoriza y se limpia el código.
- 9.Se vuelven a pasar todas las pruebas automatizadas para comprobar que todo sigue funcionando.
- Volvemos al punto 3 con los criterios de aceptación que falten y repetimos el ciclo una y otra vez hasta completar nuestra aplicación.

# 5. ¿Quién está haciendo TDD realmente?

**1. "¿Qué tan ágil es TDD?"**
Desafortunadamente, la tasa de adopción de TDD no es tan alta como debería ser realmente. La siguiente figura resume los resultados del 2010

La encuesta proporciona información sobre las estrategias de validación que siguen los equipos que dicen ser ágiles. Sospecho que las tasas de adopción informadas para el desarrollador TDD y la aceptación TDD, 53



# 6. Ventajas del TDD, Posibles problemas del TDD y Sus Soluciones

**1. Ventajas del TDD**
**- Mayor calidad**

Garantiza que las pruebas se ejecutan (no sean omitidas), evitando que las aplicaciones tengan fallas la primera vez que el usuario las ejecuta o que el usuario encuentre los errores, en lugar de ser encontrados por el equipo de desarrollo.
Asimismo, el hacer pruebas en etapas tempranas del desarrollo es una forma de incorporar la calidad al proceso, resultando en menos errores (bugs) en las etapas finales del proyecto.

**- Diseño enfocado en las necesidades**

El escribir las pruebas primero que el código, obliga a que las necesidades reales del cliente sean consideradas primero, obligando a analizar primero qué es lo que realmente se necesita que el código haga y no al contrario. Como resultado, habrá menos retrabajo después.
**- Mayor simplicidad en el diseño**

El escribir las pruebas primero que el código, obliga a que las necesidades reales del cliente sean consideradas primero, obligando a analizar primero qué es lo que realmente se necesita que el código haga y no al contrario. Como resultado, habrá menos retrabajo después.

**- Mayor simplicidad en el diseño**

Bajo TDD, en lugar de enfocarse en realizar diseños extensos y complejos, el equipo se enfocará en la necesidad o requerimiento del cliente, agregando solamente la funcionalidad que el cliente necesita. Esto es muy importante, pues es la complejidad la que produce los errores.
Esto obliga a escribir código enfocado en las necesidades del usuario, evitando antipatrones como los objetos multipropósito (clase gorda) o el acoplamiento del código, dado que desarrollarán códigos específicos a los requerimientos que se estén atendiendo en esa iteración.
**- El diseño se va adaptando al entendimiento del problema**

A medida que se realizan iteraciones de probar y programar, el entendimiento del problema se incrementa, de esta forma, sucesivas iteraciones cuentan con un mayor entendimiento, lo que reduce los malos entendidos de la funcionalidad al final del desarrollo, resultando en menos retrabajo.

**- Mayor productividad** En un proyecto tradicional, generalmente lo que sucede es que al principio se es muy productivo, sin embargo, esa productividad cae hacia el final del proyecto, cuando empiezan a encontrarse errores en todas partes, se encuentran malentendidos en lo que el cliente quería, o cuando el cliente hace un par de cambios desestabilizadores.

En contraposición a este esquema, una de las principales ventajas de TDD es que se obtiene retroalimentación (feedback) inmediato sobre el software desarrollado.
Trabajando bajo TDD, al principio se algo improductivo, pues se necesita escribir una serie de casos de prueba que fallaran al primer intento, sin embargo, los beneficios se hacen evidentes cuando se ha probado constantemente la aplicación, se han corregido los errores de forma temprana y de han aclarado de forma temprana las dudas en la funcionalidad.

**- Menos tiempo invertido en debugging de errores**

El código se va desarrollando por piezas pequeñas, por ende, cuando surge un error, los esfuerzos se enfocan en la pequeña pieza de código que fue modificada, por lo que se le pueden llegar a los problemas de forma más directa.

**2. Posibles problemas del TDD y sus soluciones**

**- Interfaz de usuario**

TDD es difícil de implementar en la capa de interfaz de usuario (presentación), debido a que está actividad contiene elementos que contienen a alargar el ciclo de prueba y desarrollo. En su lugar, TDD encaja más fácilmente en los desarrollos en las capas de lógica de negocios (objetos y dominio) y acceso a datos. Por ende, pueden existir reservas respecto a aplicar TDD en una aplicación con alto grado de interacción con el usuario.

Sin embargo, esto no es necesariamente algo malo, dado que la metodología obliga a separar las capas y a no implementar lógica de negocio en la capa de presentación, lo cual sería un anti patrón.

**- La Base de datos**

Uno de los principales problemas de usar TDD en aplicaciones con bases de datos, es que una vez que se ha ejecutado una prueba, la base de datos puede quedar en un estado distinto al que se necesita para hacer la siguiente prueba (por ejemplo, si la aplicación necesita cambiar el valor de un campo de A a B, cuando la prueba termina el valor queda en B, por lo que no se puede ejecutar una siguiente prueba).
Una forma de abordar este problema es escribir código para inicializar la base de datos en el estado previo, sin embargo esto añade carga de trabajo adicional.
Otra solución es utilizar objetos para representar la base de datos, por medio de objetos cascaron, respuestas predefinidas o dummies que emulen las respuestas de la base de datos.

**- Errores no identificados**

Sólo por el hecho de pasar todas las pruebas en la herramienta que se utilize (JUNIT por ejemplo), no significa que no se tengan errores, sólo significa que las pruebas que se han ejecutado no han encontrado errores. El utilizar TDD podría llevar a un falso sentimiento de seguridad, por ende, se necesita enfocarse en que las pruebas sean detalladas y cubran todos los escenarios posibles.

**- Perder la visión general (Ver el árbol en lugar del bosque)**

TDD es un enfoque de abajo hacia arriba (Bottom-Up), y se debe estar al tanto que podría perderse visibilidad general del proyecto y del aplicativo. Es una buena idea mantener un modelo general (bajo un enfoque tradicional como UML) y revisarlo de vez en cuanto, quizás se encuentren oportunidades para refactorizar y hacer que la aplicación se le pueda dar mantenimiento en el tiempo. **- Pronunciada curva de aprendizaje** Como se ha dicho en otras entregas, TDD es difícil de adoptar, por lo que puede esperarse un descenso en la productividad durante los primeros dos meses de implementación. Lo recomendable

para enfrentar esto es buscar ayuda, por medio de formación (cursos) y consultorías que apoyen en la adopción de la nueva forma de trabajo.

# 7. Webgrafía


- https://uniwebsidad.com/libros/tdd/capitulo-2
- https://docs.microsoft.com/es-es/previous-versions/bb932285(v=msdn.10)
- http://www.conaiisi.unsl.edu.ar/portugues/2013/158-524-1-DR.pdf