Primera Parte

1- Design Principles Behind Smalltalk – Dan Ingalls

Propósito Proyecto Smalltalk: proveer soporte de computación al espíritu creativo que todos llevamos dentro. Trabajo desarrollado a partir de una visión que incluye a:

- Un individuo creativo
- El mejor hardware de computación disponible.

Nos concentramos en dos áreas principales de investigación:

- Un lenguaje de **descripción** (lenguaje de **programación**) que sirve de interfaz entre los modelos en la mente humana y aquellos en el hardware,
- Un lenguaje de **interacción** (interfaz al **usuario**) que adapte el sistema de comunicación humano a la computadora.

Método del Trabajo:

- Construir un programa de aplicación dentro del sistema actual (hacer una observación)
- Basado en esta experiencia, rediseñar el lenguaje (construir una teoría)
- Cosntruir un nuevo sistema basado en el nuevo diseño (hacer una predicción que puede ser corroborada)

Sistema Smalltalk-80 marca nuestra quinta iteración sobre este ciclo. En este artículo, presento algunos de los **principios generales** que hemos observado en el curso de nuestro trabajo.

Empezando con un principio más social que técnico:

Dominio Personal: *Si un sistema es para servir al espíritu creativo, debe ser completamente entendible para un individuo solitario.*

El punto aquí es que el **potencial humano** se manifiesta en los individuos. Debemos proveer un **medio que pueda ser dominado completamente por un individuo**. Cualquier barrera que exista entre el usuario y alguna parte del sistema será finalmente una barrera a la expresión creativa. Cualquier parte del sistema que no pueda ser cambiada, o que no es lo suficientemente general es probablemente un **origen de impedimentos**. Si una parte del sistema funciona de manera diferente del resto, esa parte requiere un esfuerzo adicional para controlarla. Esa complicación añadida puede afectar el resultado final, e inhibir futuros esfuerzos en esa área. Podemos entonces inferir un principio general de diseño:

Buen Diseño: *Un sistema debería ser construido con un mínimo conjunto de partes no modificables; esas partes debieran ser tan generales como sea posible; y todas las partes del sistema deberían estar mantenidas en un esquema uniforme.*

Lenguaje.

Al diseñar un lenguaje para ser usado con computadoras, no es necesario mirar muy lejos para buscar indicaciones útiles. Todo lo que sabemos sobre cómo la gente piensa y se comunica es aplicable. Hay que hacer que nuestros modelos de computación sean compatibles con la mente, en vez de hacerlo al revés.

Figura 1: El alcance del diseño de un lenguaje.

La comunicación entre dos personas (o entre una persona y una computadora) incluye comunicación en dos niveles. La comunicación explícita incluye la información que es transmitida en determinado mensaje. La comunicación implícita incluye las suposiciones relevantes comunes a los dos seres.

Propósito del lenguaje: *Proveer un esquema para la comunicación.*

Comunicación explícita: las palabras y gestos expresados y percibidos.

Comunicación implícita: la cultura compartida y experiencia que forma el contexto para la comunicación explícita. En la interacción humana, mucho de la comunicación real se realiza por referencias a un contexto compartido, y el lenguaje humano está construido sobre estas alusiones. Con las computadoras pasa lo mismo.

El "cuerpo" de la computadora provee una pantalla visual de información, y percepción de la entrada del usuario humano. La "mente" de una computadora incluye los elementos de memoria y procesamiento internos y sus contenidos.

Alcance: *El diseño de un lenguaje para usar computadoras debe tratar con modelos internos, medios externos, y con la interacción entre ellos tanto en el humano como en la computadora.*

Este hecho es responsable de la dificultad de enseñar Smalltalk a gente que ve a los lenguajes de computadora en un sentido más restringido. Smalltalk no es simplemente una mejor manera de organizar procedimientos o una técnica distinta para administración de memoria. No es sólo una jerarquía extensible de tipos de datos, o una interfaz gráfica al usuario. Es todas estas cosas, y cualquier otra que sea necesaria para soportar las interacciones ilustradas en la figura.

Objetos que se Comunican

Hemos dicho que un sistema de computación debe proveer modelos que sean compatibles con los de la mente. Por lo tanto:

Objetos: *Un lenguaje de computación debe soportar el concepto de "objeto" y proveer una manera uniforme de referirse a los objetos de universo.*

El administrador de almacenamiento de Smalltalk provee un modelo orientado a objetos de la memoria de todo el sistema. La referenciación uniforme se obtiene simplemente asociando un entero que no se repite a cada objeto del sistema. Esta uniformidad es importante porque significa que las variables en el sistema pueden indicar valores ampliamente variados y aún así ser implementadas como simples posiciones de memoria. Se crean objetos al evaluarse las expresiones, y pueden ser pasados de un lado a otro gracias a la referenciación uniforme, por eso **no es necesaria una manera de almacenamiento en los procedimientos que los manipulan**. Cuando todas las referencias a un objeto han desaparecido del sistema, el propio objeto se esfuma, y su espacio de almacenamiento es recuperado. Este comportamiento es esencial para soportar completamente la metáfora de objetos.

Administración del Almacenamiento: *Para ser auténticamente "Orientado a Objetos" un sistema debe proveer administración automática del almacenamiento.*

Una manera de ver si un lenguaje está funcionando bien es ver si los programas parecen estar haciendo lo que hacen. Si están salpicados con instrucciones de administración del almacenamiento (memoria dinámica, pedir y liberar), entonces su modelo interno no está bien adaptado al de los humanos.

Mensajes: *La computación debería ser vista como una capacidad intrínseca de los objetos que pueden ser invocados uniformemente enviándoles mensajes.*

Así como los programas se ensucian si el almacenamiento de los objetos debe ser tratado explícitamente, el control en el sistema se vuelve complicado si debe ser tratado extrínsicamente.

Smalltalk provee una solución mucho más limpia: Envía el nombre de la operación deseada, juntamente con cualquier parámetro necesario, como un mensaje al número, entendiendo que el receptor es el que mejor sabe cómo realizar la operación deseada.

Metáfora Uniforme: *Un lenguaje debería ser diseñado alrededor de una metáfora poderosa que pueda ser aplicada uniformemente en todas las áreas.*

Smalltalk, está construido sobre el modelo de objetos que se comunican. En cada caso, las aplicaciones grandes son vistas en la misma manera en que las unidades fundamentales de las cuales el sistema está construido. Especialmente en Smalltalk, la interacción entre los objetos más primitivos es vista en la misma manera que la interacción de más alto nivel entre la computadora y su usuario. Todo objeto en Smalltalk, hasta un humilde entero, tiene un conjunto de mensajes, un protocolo, que define la comunicación explícita a la que ese objeto puede responder. Internamente, los objetos pueden tener almacenamiento local y acceso a otra información compartida que comprenden el contexto implícito de toda comunicación.

Organización

Una metáfora uniforme provee el marco en el cual se pueden construir sistemas complejos. Algunos principios de organización relacionados entre sí contribuyen a la administración exitosa de la complejidad. Para empezar:

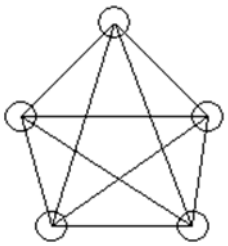

Figura 2: Complejidad de un sistema.

Al incrementarse la cantidad de componentes de un sistema, la probabilidad de interacción no deseada crece rápidamente. Por esto, un lenguaje de computadora debería ser diseñado para minimizar las posibilidades de esa interdependencia.

Modularidad: *Ningún componente en un sistema complejo debería depender de los detalles internos de ningún otro componente.*

La metáfora de envío de mensajes provee modularidad al desacoplar la intención del mensaje (incorporado en el nombre) del método usado por el receptor para realizar la intención. La información estructural es similarmente protegida porque todo acceso al estado interno de un objeto es a través de esta misma interfaz de mensajes.

La complejidad de un sistema puede muchas veces ser reducida agrupando componentes similares. Este agrupamiento es conseguido a través del tipado de datos en los lenguajes de programación convencionales, y a través de **clases** en Smalltalk. Una clase describe otros objetos -su estado interno, el protocolo de mensajes que reconocen, y los métodos internos para responder a esos mensajes. Los objetos así descriptos se llaman **instancias** de la clase. Incluso las propias clases entran en este esquema; simplemente son instancias de la clase Class, que describe el protocolo y la implementación apropiados para la descripción de los objetos:

Clasificación: *Un lenguaje debe proveer un medio para clasificar objetos similares, y para agregar nuevas clases de objetos en pie de igualdad con las clases centrales del sistema.*

La clasificación es la objetivación de la idadidad.

Las clases son el principal mecanismo de extensión en Smalltalk.

Polimorfismo: *Un programa sólo debería especificar el comportamiento esperado de los objetos, no su representación.*

Factorización: *Cada componente independiente de un sistema sólo debería aparecer en un sólo lugar.*

Ahorra tiempo, esfuerzo y espacio si los agregados al sistema sólo necesitan hacerse en un lugar. Segundo, los usuarios pueden encontrar más fácilmente un componente que satisfaga una dada necesidad. Tercero, en la ausencia de una factorización apropiada, aparecen problemas para sincronizar cambios y para asegurar que todos los componentes interdependientes son consistentes. Puedes ver que una falla en la factorización implica una violación a la modularidad.

Reaprovechamiento: *Cuando un sistema está bien factorizado, un gran reaprovechamiento está disponible tanto para los usuarios como para los implementadores.*

Los beneficios de la estructuración para los implementadores son obvios. Para empezar, habrá menos primitivas para implementar. Por ejemplo, todos los gráficos en Smalltalk se hacen con una sola operación primitiva.

Máquina Virtual: *Una especificación de máquina virtual establece un marco para la aplicación de tecnología.*

La máquina virtual de Smalltalk establece un modelo orientado a objetos para el almacenamiento, un modelo orientado a mensajes para el procesamiento, y un modelo de bitmap (mapa de bits) para el despliegue visual de información. A través del uso de microcódigo, y eventualmente hardware, la performance del sistema puede ser mejorada dramáticamente sin ningún compromiso de las otras virtudes del sistema.

Interfaz al Usuario

Una interfaz al usuario es simplemente un lenguaje en el que la mayor parte de la comunicación es visual. Dado que la presentación visual se asimila mucho a la cultura humana establecida, la estética juega un rol muy importante en esta área. Como toda la capacidad de un sistema de computación finalmente es entregada a través de la interfaz al usuario, la flexibilidad es esencial también acá. Una condición habilitante para la flexibilidad adecuada de una interfaz al usuario puede ser enunciada como un principio orientado a objetos:

Principio Reactivo: *Cada componente accesible al usuario debería ser capaz de presentarse de una manera entendible para ser observado y manipulado.*

Cada objeto provee un protocolo de mensajes apropiado para la interacción.

Debería notarse que los sistemas operativos parecen violar este principio.

Sistema Operativo: *Un sistema operativo es una colección de cosas que no encajan dentro de un lenguaje. No debería existir.*

Aquí hay algunos ejemplos de componentes de sistemas operativos convencionales que han sido incorporados naturalmente al lenguaje Smalltalk:

· **Administración del Almacenamiento** - Enteramente automático. Los objetos son creados por un mensaje a su clase y destruidos cuando no nadie los referencie. La expansión del espacio de direccionamiento mediante la memoria virtual es igualmente transparente.

· **Sistema de Archivos** - Está incorporado al esquema usual a través de objetos como Files (archivos) y Directories (directorios) con protocolos de mensajes que soportan el acceso a archivos.

· **Manejo de la Pantalla** - La pantalla es simplemente una instancia de la clase Form (figura), que es continuamente visible, y los mensajes de manipulación gráfica definidos en esa clase se usan para cambiar la imagen visible.

· **Entrada del Teclado** - Los dispositivos de entrada del usuario se modelan similarmente como objetos con mensajes apropiados para determinar su estado o leer la su historia como una secuencia de eventos.

· **Acceso a Subsistemas** - Los subsistemas se incorporan naturalmente como objetos independientes dentro de Smalltalk: aquí pueden usar el amplio universo de descripción existente, y aquellos que involucran interacción con el usuario pueden participar como componentes de la interfaz al usuario.

· **Debugger (herramienta de depuración)** - El estado del procesador de Smalltalk es accesible como una instancia de la clase Process (proceso) que es dueña de una cadena de stacks. El debugger es sólo un subsistema de Smalltalk que tiene acceso a manipular el estado de un proceso suspendido. Es de notar que casi el único error de tiempo de ejecución que puede ocurrir en Smalltalk es que un mensaje no sea entendido por su receptor.

Smalltalk no tiene "sistema operativo" como tal. Las operaciones primitivas necesarias, como leer una página del disco, son incorporadas como métodos primitivos en respuesta a mensajes Smalltalk normales.

Trabajo futuro

Selección Natural: *Los lenguajes y sistemas que son de buen diseño persistirán, sólo para ser reemplazados por otros mejores.*

2- Unit Test Guidelines

1. **Mantener los Unit Test pequeños y rápidos**

Idealmente el conjunto entero de tests debe ser ejecutado antes de cada chequeo del código. Manteniendo los tests rápidos reduce el tiempo de desarrollo.

2. **Los Unit tests deberían ser completamente automatizados y no interactivos**

El conjunto de tests es normalmente ejecutado de manera regular y debe ser completamente automático para que tenga sentido. Si los resultados requieren de inspección manual, entonces los tests no son propiamente unit tests.

3. **Hacer los Unit tests fáciles de correr**

Configurar el entorno de desarrollo para que los tests unitarios y los tests de conjunto (tests suites) puedan ser corridos con un solo comando o un solo click del mouse.

4. **Medir los tests (Measure the tests)**

Aplicar análisis de cobertura al test run para que sea posible leer con exactitud la cobertura de la ejecución e investigar cuáles partes del código son ejecutadas y cuáles no.

5. **Arreglar tests fallidos inmediatamente**

Cada desarrollador debe ser responsable de asegurarse que los nuevos tests corren de manera correcta al cheuqearlos, y que todos los tests existentes corren luego de un check in del código. Si un test falla como parte de una ejecución regular del test, el equipo entero debe dejar lo que están haciendo y asegurarse que el problema se arregle.

6. **Mantener los tests a un nivel unitario**

Unit testing trata sobre testear clases. Debería haber por lo menos un test class por cada clase ordinaria y el comportamiento de la clase debe ser testeado aisladamente. Evitar la tentación de testear un flujo entero de trabajo usando un framework (o entorno de trabajo), ya que esos tests son lentos y difíciles de mantener. El testeo del flujo de trabajo debe tener lugar, pero no es Unit Testing y debe ser configurado para correr independientemente.

7. **Comenzar de manera simple**

Un simple test es infinitamente mejor que no tener ningun test. Un simple test class va a establecer el entorno de trabajo de la clase en cuestión, va a verificar la precencia y correctitud de tanto el build environment, el ambiente de unit testing, el ambiente de ejecución y la herramienta de análisis de cobertura, y va a probar que la clase en cuestión es parte del montaje y que puede ser accedida.

8. **Mantener a los tests independientes**

Para asegurar la robustes y el simple mantenimiento, los tests nunca deberían depender de otros tests ni deberían depender del orden en que los tests se ejecuten.

9. Mantener a los tests cerca de las clases que están siendo testeadas

Si la clase a testear es Foo, el test de la clase debe llamarse FooTest (no TestFoo) y debe guardarse en el mismo paquete o directorio que Foo. Manteniendo los tests de las clases en un directorio istinto hace que sean más dificil de acceder y mantener.

Hay que asegurarse que el build environment es configurado de tal forma que los test de las clases no llegan a estar en librerias de producción o ejecutables.

10. Nombrar los tests adecuadamente

Asegurarse que cada test pruebe una particularidad distintiva de la clase que está siendo testeada, y que se llama acordemente. La convención típica es test[what] como testSaveAs(), testAddListener(), etc.

11. Testear APIs públicas

Unit testing puede ser definida como clases de testeo a través de su API pública. Algunas herramientas de testeo hacen esto posble para testear contenido privado de una clase, pero esto debería ser evitado ya que hace que el testeo sea mas verboso y difícil de mantener. Si hay contenido privado que necesita explícitamenteser testeado, considerar hacer un refactor de eso hacia métodos públicos en utility clases. Pero hacer esto para mejorar el diseño general, no para ayudar al testeo.

12. Pensar en forma de Caja Negra

Actua como una 3ra clase consumidora, y testea si la clase cumple sus requerimientos. Y trata de destruir esto.

13. Pensar en forma de Caja Blanca

Después de todo, el programador de test también escribió la clase que está siendo testeada, y esfuerzo extra debe ser puesto en testear la lógica más compleja.

14. Testear los casos triviales

A veces es recomendado que todos los casos no triviales deben ser testeados y que métodos triviales como simples setters y getters pueden ser omitidos. De todas formas, hay numerosas razones por la que estos casos deben ser testeados de también:

- Es difícil definir lo trivial. Puede significar cosas distintas para gente diferente
- Desde una perspectiva de caja negra, no siempre hay forma de saber que parte del código es o no trivial
- Los casos triviales contienen errores también, a veces como resultado de operaciones de copiar y pegar.
- La recomendación entonces es testear todo. Después de todo, los casos triviales son fáciles de testear.

15. Enfocarse primero en cobertura de ejecución

Distinguir entre cobertura de ejecución y verdadera cobertura de test. El objetivo inicial del testeo debe ser de asegurar una alta cobertura de ejecución. Esto va a asegurar que el código es efectivamente ejecutado en algunos parámetros de entrada. Cuando esto está en lugar, la cobertura del test debe mejorarse. Notar que la cobertura efectiva del test no puede ser facilmente medida. Considerar el siguiente ejemplo:

```
void setLength(double length);
```

Llamano a setLength(1.0) capaz tenes 100% de cobertura de ejecución. Para alcanzar el 100% de cobertura real, el método debería llamarse para todo valor posible y esto no es posible.

16. Cubrir casos borde

Hay que asegurarse que los parámetros con casos bordes son cubiertos. Para números, testeo con negativos, 0, números chicos, grandes, NaN, infinito, etc. Para Strings, un carácter solo, non-ASCII, etc. Para colecciones, testeo con vacías, y así con todo. La clase testeado va a sugerir el caso borde en particular dependiendo lo que use. El punto de esto es de asegurarse que los candidatos de error son previamente testeados.

17. Proveer de un generador random

Cuando los casos bordes están cubiertos, una manera simple de mejorar la cobertura aún más es generar parámetros de manera aleatoria asi los tests pueden ejecutarse con una entrada diferente cada vez.

18. Testear cada característica una vez

Cuando estamos en testing mode a veces es tentador realizar asserts sobre todo en cada test. Esto debería evitarse ya que es difícil de mantener. Testear exactamente la característica indicada en el nombre del test. En cuanto a código, el objetivo es el de mantener la menor catidad de código posible.

19. Usar asserts explícitos

Siempre es preferible usar assertEquals(a,b) a assertTrue(a == b). Esto es particularmente importante en la combinación de parámetros con valores random como fue descrito más arriba.

20. Proveer tests negativos

Tests intencionalmente negativos verifican la robustes y manejor apropiado de errores.

21. Diseñar el código con el testeo en mente

Escribir y mantener tests unitarios es costozo, y minimizando la API pública y reduciendo la complejidad cíclica en el código son maneras de reducir este costo y conseguir una mayor cobertura de test, más fácil de escribir y mantener.

22. No conectar con recursos externos predefinidos

Los tests unitarios deben ser escritos sin conocimiento explícito del contexto del ambiente en el que van a ser ejecutados para que ellos puedan ser corridos en cualquier lugar y en cualquier momento. En función de proveer los recursos requeridos para el test, estos recursos en lugar deben hacerse dispoibles por el test mismo.

Considerar por ejemplo una lase para parsear archivos de un cierto tipo. En lugar de recoger una muestra de ejemplo de una ubicación predefinida, hay que poner un archivo de contenido dentro del test, escribirlo en un archivo temporal en el setup del test y luego eliminarlo cuando el test se haya completado.

23. Conocer el costo del testing

No escribir unit tests es costoso, pero escribirlos es costoso también. Hay un intercambio entre estos dos, y en términos de cobertura de ejecución, el estándar típico de la industria es aproximadamente un 80%.

24. Priorizar el testing

Unit testing es tipicamente la parte inferior de un proceso, y si no hay suficientes recursos para testear todas las partes del sistema, la prioridad debe enfocarse en las capas más bajas primero.

25. Preparar test code para errores

26. Escribir tests para reproducir bugs

Cuando se reporta un bug, escribir un test para reproducirlo (un failing test) y usar este test como criterio para arreglar el código.

27. Mantenerlo sencillo

Unit tests deben ser simples para ser efectivas, no deben contener lógica de comprensión n si mismos. Algo qu genere ruido es si el unit test está duplicando alguna lógica del código siendo testeado, o si parece que el código de prueba necesita a su vez testearse también.

28. Conocer las limitaciones

Unit tests nunca pueden probar que el código sea completamente correcto. Un test fallido puede indicar que el código contiene errores, pero un test logrado no es prueba de nada.

La aplicación más útil de los unit tests es la verificación y documentación de los requerimentos a bajo nivel, y el testeo de regresión: verificando que las invariantes del código permanecen estables a lo largo de la evolución y refactoring del código.

Consecuentemente unit tests nunca pueden reemplazar un diseño up-front. Unit tests deben ser usados como suplementos valiosos de las metodologías establecidas de desarrollo. Y tal vez lo más importante: el uso de unit tests fuerza a los desarrolladores a pensar a través de sus diseños, lo cual por lo general mejora la calidad del código y de las API´s.

3-What is the Point of Test-Driven Development?

Sofwtare Development as a Learning Process

Feedback is the Fundamental tool:

El mejor acercamiento que un equipo puede tomar es el uso de **feedback empírico** para aprender acerca del sistema y su uso, y luego aplicar ese conocimiento de vuelta al sistema. Aplicar ciclos de feedback luego de cada nivel de despliegue.

En un proyecto organizado como un conjunto anidado de ciclos de feedback, el desarrollo es **incremental** e **iterativo.**

Desarrollo incremental construye un sistema característica por característica, en lugar de construir todas las capas y componentes e integrandolas a todas al final. Cada característica es implementada como una porción end-to-end a través de todas las partes relevantes del sistema. El sistema siempre está integrado y listo para su despliegue.

Desarrollo iterativo prograsivamente refina la implementación de las características en respuesta del feedback hasta que sean lo suficientemente buenas.

Prácticas que apoyan el cambio:

Necesitamos dos fundamentos técnicos si queremos crecer un sistema confiable y atento a cambios anticipados que siempre pasan:

- Primero, necesitamos testing constante para atrapar errores de regresión, para que podamos agregar nuevas características sin romper las existentes previas.
- Segundo, necesitamos mantener el código lo más simple posible, para que sea más fácil de entender y modificar. Estamos haciendo refactor constantemente.

Test Driven Development: escribimos las pruebas antes del código, en lugar de simplemente usar pruebas para verificar nuestro trabajo luego de que esté hecho. TDD da feedback de la calidad tanto de la **implementación** como del **diseño.** Escribir tests:

- Nos hace mas claro el criterio de aceptación para la próxima parte de trabajo – nos tenemos que preguntar cómo poemos saber cuando hayamos terminado el diseño.
- Nos alienta a escribir componentes perdidamente, para que estos puedan ser testeados aisladamente y luego combinados a más alto nivel con otros
- Agrega una descripción ejecutable sobre lo que hace el código
- Se agrega a un conjunto completo de regresión

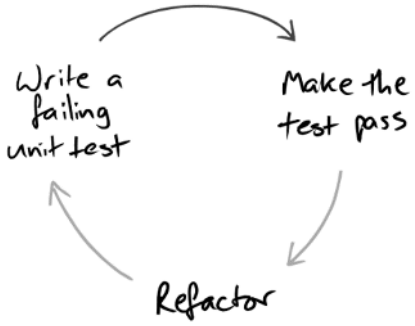

Figure 1.1 *The fundamental TDD cycle*

Los running tests:

- Detectan errores mientras el contexto se encuentra fresco en nuestra mente
- Nos hacen saber cuando hemos hecho lo suficiente, desalentando partes y características innecesarias del diseño.

TDD Golden Rule: Refactoring. Think Local, Act Local

Refactoring*:* cambiar la estructura interna de un cuerpo existente de código sin cambiar su comportamiento. El programador aplica unas series de transformaciones o refactorings que no cambian el comportamiento del código. El programador se aseguro que el sistema todavía funciona luego de cada paso.

The Bigger Picture

La regla dorada nos dice lo que tenemos que hacer: *Escribir un test fallido.* Comenzamos por escribir un test de aceptación (acceptance test).

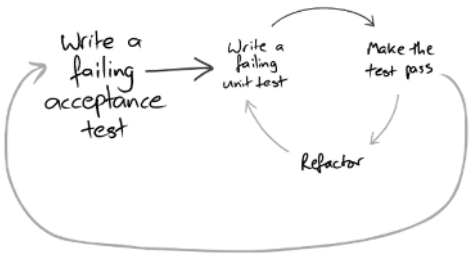

Figure 1.2 *Inner and outer feedback loops in TDD*

Testing End-to-End

Un test end-to-end interactua con el sistema solo desde afuera: a través de la interfaz de usuario, nviando mensajes como si fuese un 3ro del sistema.

Para nosotros end-to-end significa más que interactuar con un sistema desde fuera. Preferimos tener end-to-end tests que prueben tanto al sistema como al proceso por el cual es construido y desplegado. Un build automático, usualmente hecho por alguien que esté chequeando el código dentro del repositorio fuente va a:

- Chequear la última versión
- Compilar y hacer unit-test del código
- Integrar y empaquetar al sistema
- Realizar un despliegue estilo producción en un ambiente real
- Ejercitar al sistema a través de sus puntos de acceso externos.
- Check out the latest version

Un sistema es *desplegable (deployable)* cuando todos los acceptance tests pasan, ya que ellos deberían darnos la confianza suficiente de que todo funciona.

Levels of Testing

Jerarquía de los tests:

Acceptance: Todo el sistema funciona?

Integration: Nuestro código funciona frente a código que no podemos cambiar?

Unit: Nuestros objetos hacen las cosas correctamente y son convenientes para trabajar?

En algunos casos, los tests de aceptación pueden no ser end-to-end.

Usamos el término **integración (integration)** para referirnos a los tests que chequean como funciona nuestro código con código por afuera del equipo de trabajo que no podemos cambiar.

External and Internal Quality

Podemos hacer una distinción entre calidad externa e interna:

- *Calidad externa (External quality):* es cómo nuestro sistema cumple con las necesidades de los clientes o usuarios.
- *Calidad interna (Internal quality):* es que tan bien nuestro sistema umple con las necesidades de sus desarolladores y administradores.

Correr un end-to-end test nos habla acerca de la calidad externa de nuestro sistema, y escribiendo estos tests nos dice que tan bien nosotros (el equipo entero) entendemos el dominio. Escribir unit tests nos da mucho feedback

acerca de la calidad de nuestro código, y correrlos nos dice que no hemos roto ninguna de nuestras clases.

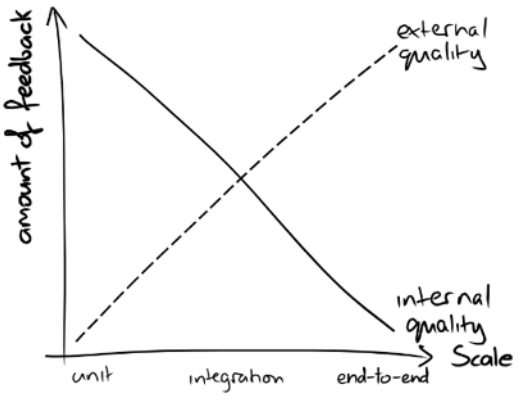

Figure 1.3 *Feedback from tests*

Coupling and Cohesion

Estas son métricas que (dificilmente) describen que tan fácil es intercambiar el comportamiento de cierto código.

Elementos que están acoplados (coupled) si uno cambia fuerza un cambio en el otro.

La cohesión de un elemento es una medida de si las responsabilidades de este forman una unidad significativa.

4-8 Principles of Better Unit Testing

Las reglas que aplican para escribir buen código de producción no siempren aplican a crear un buen unit test.

¿Qué hace a una buena unit test?

Las unit test son cortas, rápidas y automatizadas y aseguran que una parte específica de tu programa funciona. Testean una funcionalidad específica de un método o clase. Un buen unit test sigue estas reglas:

- El test falla solo si un nuevo bug es introducido al sistema o cambian los requerimentos.
- Cuando el test falla, es facil de entender la razón por la que lo hace.

Para escribir buenas pruebas unitarias, se deben seguir estas pautas:

1. Sepa lo que está probando: Una prueba escrita sin un objetivo claro es fácil de detectar (larga, difícil de entender, evalúa mas de una cosa). Un truco es utilizar el escenario probado y el resutado esperado como parte del nombre del método de prueba. Probar solo una cosa hace que sea más legible y es más facil encontrar la causa si falla.

2. Las pruebas unitarias deben ser autosuficientes: Una buena prueba unitaria debe estar aislada. Una sola prueba no debería depender de ejecutar otras pruebas antes ni del orden de ejecución. Ejecutarla 1000 veces debería dar siempre el mismo resultado. El uso de estados globales puede causar "fugas" entre las pruebas, por eso hay que asegurarse de iniciarlos correctamente y de limpiar cada uno de estos estados globales entre corridas de pruebas o evitar usarlos directamente.

3. Las pruebas deben ser deterministas: Una prueba debe pasar todo el tiempo o fallar hasta que se solucione. Otra "práctica" que debe evitarse es escribir pruebas con entrada aleatoria ya que introduce incertidumbre.

4. Convenciones de nomenclatura: Lo primero que nota sobre una prueba fallida es su nombre. Cuando falla una prueba bien nombrada es más fácil entender qué se probó y por qué falló. Por ejemplo, para una clase Calculadora que puede dividir dos números un buen nombre de test es Dividir_DividirPorCero_LanzaExcepcion().

5. Repítete: En las pruebas unitarias es muy importante la legibilidad por lo que es aceptable el código duplicado. Esto no es así en código de producción. En este caso de pruebas, eliminar duplicados estaría bien solo si no oscurece nada.

6. Prueba resultados, no implementación: Las pruebas unitarias exitosas solo fallarían en caso de un error real o un cambio de requisitos. Evitar escribir pruebas frágiles, ya que fallarían debido a cambios internos en el software que no afectan al usuario. Solo pruebe métodos privados si tiene una muy buena razón. La refactorización trivial puede causar errores complicados y fallas en las pruebas.

7. Evitar especificaciones excesivas: Hay que evitar escribir tests muy específicos, como por ejemplo testear que un método fue llamado 'x' cantidad de veces. Use un marco de aislamiento para establecer el comportamiento predeterminado de los objetos externos y asegurarse de que no esté configurado para lanzar excepción si se llamó a un método inesperado.

8. Utilice un marco de aislamiento: Un marco de aislamiento es una biblioteca de terceros y un gran ahorro de tiempo. Los ahorros en líneas de código entre el uso de un marco de aislamiento y la escritura de simulaciones a mano para el mismo código pueden llegar al 90%. Cada marco tiene un conjunto de API para crear y usar objetos falsos sin que el usuario necesite mantener detalles irrelevantes de la prueba específica. Si se crea una falsificación para una clase específica, cuando esa clase agrega un nuevo método, no es necesario cambiar nada en la prueba

5-The Art of Enbugging

El hecho de llamar a los errores bugs hace pensar que aparecen espontaneamente en el código cuando en realidad no es así. Los programadores los ponen ahí. En un mal día se puede sentir como que estás *enbugging* el código. Lo que suele pasar no es que se ponen directamente, sino que se setean las condiciones para que luego pongamos los bugs. Una forma de prevenir esto es mantener una separación de intereses (*separation of concerns*), que significa diseñar el código para que las clases y módulos tengan responsabilidades claras, bien definidas y aisladas, y semánticas entendibles.

Es complicado pero una recomendación es escribir *shy code*, código que no revela mucho de si mismo.

Tell, don't ask

En Java y C++ el hecho de que la invocación de un método se parece mucho a la llamada de una función nos distrae de la metáfora de mensajes.

El código de procedimiento tiende a obtener información y luego tomar decisiones basadas en ella. El código orientado a objetos les dice a los objetos que hagan cosas. Explícitamente NO queremos preguntar a un objeto acerca de su estado, hacer una decisión y luego decirle que hacer.

La lógica que está implementando es probablemente responsabilidad del objeto llamado, no suya como persona que llama. Tomar decisiones fuera del objeto viola su encapsulación y tiende a generar bugs. Queremos decirles a los objetos qué hacer, no preguntarles su estado.

Que la persona que llama no sepa demasiado sobre cómo se ejecutará su comando significa que hemos reducido el acoplamiento. Los métodos de consulta no deben tener efectos secundarios, solo brindar información sobre el estado del objeto que puede servir, por ejemplo, para los tests.

• También se desea hablar con la menor cantidad posible de objetos ya que sino tenderá más a romperse. Una herramienta útil para este problema se llama "Ley de Demeter para funciones". Esta buena idea sugiere que un objeto solo debería llamar:

The Pretty Good Idea of Demeter

Mientras a más objetos les hables, mayor será el riesgo de rompimiento cuando uno de esos objetos cambia. Una buena herramiento a aplicar es la llamada "Law of Demeter for Functions" (dogmático llamarlo ley, sería mejor llamrla una "buena idea"), que sugiere que un objeto debería solo llamar a:

- A sí mismo

- Cualquier parámetro pasado al método.

- Cualquier objeto que este creó.

- Cualquier objeto directamente contenido dentro de este.

Para desligarse de responsabilidad entre objetos, hay veces que se terminan usando muchos wrappers pequeños que no hacen nada más que delegar contenido. Esto trae un costo de ineficiencia en contrapuesta a un mayor acoplamiento.

Programar cerca del dominio del problema.

A largo plazo, el acoplamiento de clase superior es inaceptable, ya que aumenta las posibilidades de que cualquier cambio que se realice rompa algo en otro lugar.

En aquellos casos en las que la velocidad es primordial y el acoplamiento alto es aceptable, acóplarlo al máximo. Dejar claro en la documentación que estas clases particulares están unidas entre sí y por qué

6- Getter Erradicator – Martin Fowler

La justificación general para eliminar getters es la violación del encapsulamiento. Agregar getters es ligeramente mejor a hacer los campos públicos. El autor recomienda no escribir los getters hasta que realmente los necesites, pero también trae la peligrosidad de perder el punto del encapsulamiento.

El autor dice que el fin del encapsulamiento no es el de ocultar data sino decisiones, particularmente en los lugares donde pueden cambiar. La pregunta para hacerse es "¿Qué pieza de variablidad estás ocultando y por qué?" en lugar de "¿estoy exponiendo data?".

Se cree que la verdadera motivación para la eliminación de los getters es que mucho del código de los lenguajes OO sigue teniendo muchas cosas de procedural que scan data de un objeto para hacer algo. Esto se puede lograr solo usando getters por lo que es entendible que se recomiende no usarlos, pero puede ser una herramienta no muy buena el usar esto para lograr el objetivo. Hay muchas veces que el uso de getters es justificado.

Una alerta de mal uso de getters es la de Data Class, una clase que solo tiene campos y getters. En estos casos puede ser útil el preguntarse "¿puedo deshacerme de este getter?".

Poner el comportamiento en la misma clase que la data es lo que Craig Larman llama "Information Expert". Una buena regla es que las cosas que cambian juntas deben estar juntas.

7-Replace Conditional with Polymorphism

Problema: Tiene un condicional que realiza varias acciones según el tipo de objeto o las propiedades.

Solución: Crear subclases que coincidan con las ramas del condicional. En ellos crear un método compartido y mover el código de la rama correspondiente del condicional a él. Luego reemplazar el condicional con la llamada al método. (Polimorfismo).

¿Por qué hacer Refactor?

Esta técnica de refactorización puede ayudar si el código contiene operadores realizando varias tareas basadas en:

- La clase del objecto o la interfaz que implementa
- El valor de un campo del objeto
- El resultado de llamar a uno de los métodos del objeto

Beneficios

- Esta técnica adhiere al principio de *Tell don´t Ask,* en lugar de preguntarle a un objeto acerca de su estado, este realiza acciones basado en esto.
- Remueve el código duplicado
- Si necesitas agregar una nueva variante de ejecución, lo único que tenes que hacer es agregar una nueva subclase sin tocar el código existente (Open/Closed Principle).

¿Cómo hacer Refactor?

Hay que tener una jerarquía de clases. Otras técnicas son:

- **Reemplazar Type Code con subclases:** las subclases van a ser creadas para todos los valores de una propiedad de un objeto. Este acercamiento es simple pero menos flexibles ya que no podes crear sublcases para las otras propiedades del objeto
- **Reemplazar Type Code con State/Strategy:** una clase va a ser dedicada a una propiedad particular de un objeto, y subclases van a ser creadas desde este para cada valor de la propiedad. La clase actual va a contener referencias a objetos de este tipo y delegarles la ejecución a ellos.

Pasos del refactor

1. Si el condicional está en un método que realiza otras acciones, realizar un **Extract Method**
2. Por cada subclase jerarquía, redefinir el método que contiene el condicional y copiar el código de la correspondiente rama a esa ubicación

3. Eliminar esta rama del condicional
4. Repetir hasta que el condicional se encuentre vacío. Luego eliminar el condicional y declarar el método abstracto.

8- ¿Para qué sirve un modelo? – Martin Fowler

El crear un diagrama involucra un costo y no es algo fundamental para el cliente. El modelo no tiene valor por sí mismo. El valor que provee el modelo es darle al usuario un entendimiento de algo que sea mayor que el software mismo.
Cuando tenes un modelo muy detallado debes preguntarte cuánto estás ganando. A menudo se pierde la claridad del flujo de control. Otra función importante de los modelos es resaltar en lugar de sumergirse en todos los detalles. La cuestión es poder entender primero las estructuras claves que hacen que el software funcione y luego estar en una mejor posición para investigar todos los detalles.
Para el autor un modelo esquelético es mejor que un modelo de cuerpo entero. Es más fácil de mantener que el otro y los detalles importantes cambian con menos frecuencia. Además, tiende a estar más actualizado y sintonizado con el código.
En el de cuerpo entero o incluyes todos los detalles o decides dejar algo afuera. Tan pronto como dejes algo afuera estarás yendo por el camina esquelético ya sea más o menos anoréxico un diagrama de otro.
CASE: herramienta que mantiene un vínculo automatizado con el código. Una herramienta no puede diferenciar detalles importantes de asuntos secundarios, por lo que el resultado es siempre de cuerpo entero y además trabajan no con interfaces, sino con estructuras de datos internas que tal vez deberían estar ocultas.
Para poder ver la madera de los árboles en el diseño, necesitas un modelo esquelético, pero ni bien hagas esto perdés habilidad de tener una conexión automatizada con los detalles.

9- Pruebas de Software – Carlos Fontela

- La **verificación** tiene que ver con controlar que hayamos construido el producto tal como pretendimos construirlo.
- La **validación** controla que hayamos construido el producto que nuestro **cliente** quería.

Hay pruebas centradas en probar funcionalidades (ej.: Si intento colocar una ficha en una celda ocupada, el programa debe impedirlo). Estas son **PRUEBAS FUNCIONALES**. Pero a veces hay que probar características del sistema que no son funcionales (ej.: El tiempo de respuesta del juego no puede ser superior a 1 segundo). Estas son **PRUEBAS DE ATRIBUTOS DE CALIDAD.**

Algunos tipos de pruebas de atributos de calidad son:

- o Pruebas de compatibilidad: chequean diferentes configuraciones de hardware o de red y de plataformas de software que debe soportar el producto.
- o Pruebas de rendimiento (en condiciones de uso habitual)
- o Pruebas de resistencia o de estrés: comprueban comportamiento del sistema ante demandas extremas de recursos.
- o Pruebas de seguridad
- o Pruebas de recuperación (luego de una falla)
- o Pruebas de instalación

Alcance de las pruebas

Alcance de las pruebas de verificación (o técnicas)

Las pruebas de verificación se pueden clasificar en pruebas unitarias o de integración:

- Las **pruebas unitarias** verifican pequeñas porciones del código, alguna responsabilidad de algún método, como su postcondición. Son pruebas que ejecutan los programadores para ver que lo que acaban de escribir hace lo que ellos pretendían.
- Las **pruebas de integración** prueban que varias porciones de código, trabajando en conjunto hacen lo que pretendíamos. En POO incluyen varios métodos, clases o inclusos subsistemas enteros.

Ambos tipos de pruebas pueden ser automatizadas mediante herramientas dispoinbles para cada lenguaje y plataforma. Las herramientas en cuestión

suelen ser frameworks xUnit (llamados asi porque pueden ser SUnit, JUnit, etc).

En ambos casos, las pruebas de verificación se pueden y suelen escribir antes del código que prueban.

Las **pruebas de integración** no siempre prueban el sistema como un todo, sino que prueban partes que se quieren aislar de otras. Las pruebas de verificación podrían ser de caja negra o de caja blanca.

Prueba de caja negra cuando la ejecutamos sin mirar el código que estamos probando. **Caja blanca** es cuando se analiza el código. En general se prefiere hacer pruebas de caja negra, precisamente porque se desea probar el funcionamiento y no verificar calidad del código. ***Debugging*** es una técnica de **caja blanca**.

Prueba de escritorio: de caja blanca, el programador da valores a ciertas variables y recorre mentalmente el código ayudandose con anotaciones en un papel para ver si se comporta como él espera. El problema es que el programador puede estar sesgado por sus propios errores.

Revisiones de código: realizadas de a dos o más programadores.

Alcance de las pruebas de validación (o de usuarios)

Parecería que las deben ejecutar los usuarios, pero hay alternativas. Se suele trabajar con pruebas de aceptación diseñadas por usuarios, o diseñadas por el equipo de desarrollo y validadas por usuarios pero que ejecuta el equipo de desarrollo. Estas pruebas se suelen denominar **pruebas de aceptación de usuarios (UAT)**. Se deben ejecutar en un entorno lo más parecido a producción.

Pruebas alfa: si las pruebas se hacen en un entorno controlado por el equipo de desarrollo.

Pruebas beta: el producto se deja a disposición del cliente para que lo pruebe en su entorno.

Pruebas de comportamiento: en ocasiones en que se desa probar solamente comportamiento pero sin interfazz de usuario.

Pruebas en producción

Parece un enfoque deshonesto, pero si los errores no causan grandes riesgos podría ser una opción.

La mayor cantidad de pruebas que debe haber son las unitarias, y las de menos a través de la interfaz de usuario. Esto es porque las unitarias se ejecutan más rapidamente que las de integración, las cuales son más rápidas que las de comportamiento y estas son más rápidas que las e aceptación.

Las pruebas de aceptación son más frágiles que las de comportamiento, pues dependen de la interfaz de usuario que suele ser cambiante.

Debe haber pruebas de todos los niveles.

Roles del desarrollo ante las pruebas

Visiones tradicionales: se sostiene que debe haber personas con el rol de programadores y otros con el rol de testers. La visión tradicional separa roles y responsabilidades en dos equipos distintos dentro del desarrollo.

Visión ágil: varios métodos ágiles, entre ellos Extreme Programming y Scrum plantearon que todo el equipo de desarrollo debe trabajar de consuo y en aras de entregar un producto de calidad, por lo que, la responsabilidad de entregar un producto de calidad y sin errores pasó a ser de todo el equipo de desarrollo (programadores y testers).

Scrum plantea que no debe haber roles diferentes dentro del equipo: todos son parte del Scrum Delivery Team y no hay distinción ni siquiera para el Scrum Master. Visto desde afuera todos son *desarrolladores*. Lo que se destaca también en los métodos ágiles es que, aún cuando haya roles diferentes de programadores y testers, deben trabajar juntos, al menos para que el programador pueda comprender la perspectiva del tester, aprenda a programar código más sencillo de probar y el tester pueda apreder a automatizar pruebas.

Pruebas automatizadas: quién las desarrolla

Las **pruebas unitarias** las debe escribir y ejecutar el programador.

Pruebas de integración técnicas también son pruebas de programador: conviene que las escriban los programadores y que las ejecuten sobre servidores de integración.

Pruebas de comportamiento están a mitad de camino entre pruebas de programadores y pruebas de testers, por lo que cualquiera de los dos roles podría desarrollarlas.

Con las **UAT automatizadas** se pretende que las escriban usuarios o analistas de negocio. Si esto no se consiguiera las podrían escribir los testers pero es imperativo validarlas con los usuarios para no construir algo diferente a lo que ellos desean.

Pruebas manuales: quién diseña la prueba

Se suelen dejar como pruebas manuales algunos tipos de UAT que luego ejecutarán testers y clientes.

¿Cuándo probamos?

Los métoos ágiles abogaron por la prueba más o menos continua, automatizando todo lo posible y haciendo que las pruebas guiaran el proceso de desarrollo.

Cada vez que se agregan características nuevas a un programa, podemos provocar que dejen de funcionar cosas que ya habían sido probadas y entregadas: esto es lo que se llama una **regresión**. Para evitarlas, cada tanto se ejecutan **pruebas de regresión** que no es otra cosa que ejcutar las pruebas de todo el sistema a intervalos regulares.

Ventajas de la automatización

- Nos independizamos del factor humano, la subjetividad y variablidad en el tiempo
- Es más fácil repetir las mismas pruebas con un costo ínfimo. Esto es aplicable a regresiones, debugging y errores provenientes del sistema ya en producción.
- Las pruebas en códig sirven como herramienta de comunicación.

TDD

3 sub-prácticas

- Automatización
- Test-First: las pruebas se escriben antes del propio código a probar
- Refactorización posterior: para mantener la calidad del código, se lo cambia sin cambiar la funcionalidad, manteniendo las pruebas como reaseguro.

Ventajas del TDD:

- Las pruebas sirven como documentación del uso esperado
- Las pruebas ayudan a entender mejor lo que se está por desarrollar y suelen incluir más asos de pruebas negativas que las que escribimos a posteriori
- Escribir las pruebas antes del código minimiza el condicionamiento del autor por lo ya construido.

- Permite especificar el comportamiento sin restringirse a una única implementación.

Integración y entregas continuas

Integración continua: la idea es facilitar y automatizar al máximo las integraciones antes de liberar el producto que se está desarrollando. Consiste en realizar la compilación, construcción y pruebas del producto en forma sucesiva y automática como parte de la integración.

Entrega continua: derivación de la integración continua. Este pretende que el código siempre esté en condiciones de ser desplegado en el ambiente productivo. Se puede ser más laxo en cuanto a la frecuencia de las ejecuciones de las pruebas de aceptación.

Despliegue continuo: no solo pretende que se esté permanentemente en condiciones de desplegar, sino que efectivamente, cada cambio se despliegue en el ambiente de producción. Debido a la necesidad de desplegar en forma tan frecuente, suele descansar en ejecuciones sistemáticas de pruebas en producción.

Con estas prácticas se disminuye el riesgo de la aparición de errores en las pruebas, porque cualquier problema que surja es atribuible al último tramo de código desarrollado e integrado.

¿Cómo se diseña una prueba?

Diseño de pruebas unitarias y técnicas en general

Pruebas de caja negra. Primera forma es ponernos en el rol de quien usa el módulo a probar, que en el caso de las pruebas unitarias es el mismo programador. Podemos establecer precondiciones y postcondiciones.

Diseño de pruebas de cliente

Si la prueba debe chequear un requisito del cliente, es bueno usar la técnica de especificar con ejemplos, para los cuáles el rol del programador suele ser el de quien puede prever las clases de equivalencia y valores especiales.

10- Continuous Integration – Martin Fowler

Originada con el Extreme Programming, es una práctica de desarrollo de software donde miembros del equipo integran su trabajo frecuentemente, por lo general cada persona lo hace al menos diariamente, llevando a muchas integraciones por día. Cada integración es verificada por un build automático incluyendo tests para detectar errores de integración lo más rápido posible. Muchos equipos encuentran que este acercamiento lleva a una reducción significativa de problemas de integración y le permiten al equipo desarrollar software cohesivo más rapidamente.

Es una práctica común usar un Continuous integration server.

Agregar una característica con CI

Se comienza tomando una copia de la fuente actual de integración en la máquina local.

Se toma la copia de trabajo y se hace lo que haga falta para completar la tarea (production adding code or changing automatd tests, etc).

Una vez se finaliza, se lleva a un build automática en la máquina local, que lo compila y corre las pruebas. Solo si corren todas las pruebas sin error puede considerarse bueno.

Se hace un commit de los cambios al repositorio, pero antes se realiza un update por si alguien realizó cambios en el medio y se hace un rebuild antes del commit.

Una vez realizado esto último, no termina acá. En este punto se realiza otro build pero este tiempo en una máquina de integración en la main line del código. Solo si este build es correcto se da por finalizado.

Prácticas de la Continuous Integration

- **Mantener un solo repositorio fuente**

Todo archivo necesario para el build debe estar ahí. Se debe poder correr el código con una máquina virgen con acceso al repositorio (algunas cosas como el sistema operativo, el ambiente de java y cosas dificiles de instalar no hacen falta).

- **Automatizar el build**

Hay que asegurarse que se puede hacer el build y el launch del sistema usando un solo comando. “Cualquiera con una máquina virgen debe poder tomar las fuentes del repositorio, realizar un solo comando y tener un sistema andando en su máquina.

- **Hacer tu build auto-testing**

Para que un build sea auto-testing (self-testing), el fallo de un test debe causar el fallo del build.

- **Todos hacen commits a la Main line todos los días**

Se hace para que no pase mucho tiempo entre commits y así evitar errores ya que es más fácil de resolver en caso de que haya un bug. En la práctica es mas útil si los desarrolladores hacen más de un commit por día. Mientras más frecuentemente haces un commit, menos espacio hay para conflictos o errores, y además hace que los desarrolladores partan su código en pequeños pedazos.

- **Todo commit debe ser construido en la Main line en una máquina de Integración**

- **Arregla Broken Builds inmediatamente**

- **Mantener el Build Rápido**

El punto del continuous integration es el de proveer un feedback rápido, y no hay nada peor que un largo tiempo de build. Para la mayoría de los procesos, la guía de XP dice que 10 minutos está perfecto.

- **Testear en un clon del ambiente de producción**

Es con el objetivo de desechar cualquier problema que puede tener el sistema en producción. Por eso hay que probar en diferentes ambientes.

- **Hacer que sea fácil para cualquiera el obtener el último ejecutable**

- **Que cualquiera pueda ver lo que está sucediendo**

- **Automatizar el despliegue (automate deployment)**

11 - Estado del arte y tendencias en Test-Driven Development - Carlos Fontela

Introducción

Definición: práctica de diseño de software orientada a objetos.

TDD como práctica de metodología:

Incluye tres sub-prácticas:

- Automatización: las pruebas del programa deben ser hechas en código, y con la simple corrida del código de pruebas debemos saber si lo que estamos probando funciona bien o mal.
- Test-First: las pruebas se escriben antes del propio código a probar.
- Refactorización posterior: para mantener la calidad del diseño, se cambia el diseño sin cambiar la funcionalidad, manteniendo las pruebas como reaseguro.

Ventajas:

- Automatización:
  - Independización del factor humano, con su carga de subjetividad y variabilidad en el tiempo.
  - Repetición de las mismas pruebas, con un costo ínfimo comparado con las pruebas realizadas por una persona
- Test-first:
  - Da confianza de que el código que uno escribe siempre funciona.
  - Permite especificar el comportamiento sin restringirse a una única implementación
- Refactor:
  - Facilita el mantenimiento de un buen diseño.

Reglas:

"Nunca escribas una nueva funcionalidad sin una prueba que falle antes."

"Si no puedes escribir una prueba para lo que estás por codear, entonces no deberías estar pensando en codearlo."

La aparición de los frameworks y el corrimiento de TDD a UTDD

Problema: TDD se vio como sinónimo de pruebas unitarias automatizadas realizadas antes de escribir el código.

Causas de esta visión:

1. Primer causa: proviene del propio nombre de TDD que incluye la palabra "test". Además, las herramientas que surgieron de este método obligaban a empezar las pruebas con "test" o requerían que derivasen de la clase "TestCase".
2. Segunda causa: en todos los ejemplos que plantean los creadores, se trabaja sobre pequeñas porciones de código e incluso ellos mismos hablan de pruebas unitarias.

Ventajas de UTDD:

- Las pruebas en código sirven como documentación.

- ❖ Las pruebas en código indican con menor ambigüedad lo que las clases y métodos deben hacer.
- ❖ Las pruebas escritas con anterioridad ayudan a entender mejor la clase que se está creando.
- ❖ Las pruebas escritas con anterioridad suelen incluir más casos de pruebas negativas.

Desventajas de UTDD:

- ❖ Tiende a basar todo el desarrollo en la programación de pequeñas unidades, sin una visión del conjunto.
- ❖ Muchos critican que la arquitectura evolucione sola sin hacer nada de diseño previo.
- ❖ No permite probar interfaces ni comportamiento esperado por el cliente.
- ❖ Los cambios de diseño medianos y grandes suelen exigir cambios en las pruebas unitarias.

Los primeros intentos de pruebas de integración

Problema de la integración: qué desarrollar primero y cómo probar las interacciones con módulos aún no implementados.
Solución: la idea más simple es la de construir módulos ficticios o stubs.

Objetos ficticios:

- ❖ Dummy object (objeto ficticio): aquellos objetos que se generan para probar una funcionalidad, pero no se utilizan en la prueba.
  - ➢ Ejemplo: cuando un método necesita un objeto como parámetro, pero éste no se usa en la prueba.
- ❖ Test Stub (muñón): son los que reemplazan a objetos reales del sistema, generalmente para generar entradas de datos o impulsar funcionalidades del objeto que está siendo probado.
  - ➢ Ejemplo: objetos que invocan mensajes sobre el objeto sobetido a prueba.
- ❖ Test Spy (espía): se usan para verificar los mensajes que envía el objeto que se está probando, una vez corrida la prueba.
- ❖ Mock Object (objeto de imitación): son objetos que reemplazan a objetos reales para observar los mensajes enviados a otros objetos.
- ❖ Fake Objetct (objeto falso): reemplazan objetos con una implementación alternativa.

Objetivos:

- ❖ Disminuir las dependencias.
- ❖ Mantener las pruebas de integración como pruebas unitarias.

El punto de vista de los requerimientos del comportamiento:

Otras prácticas de diseño:

- ATDD: obtiene el producto a partir de pruebas de aceptación.
- BTDD: TDD bien hecho.
- STDD: utiliza pruebas como ejemplos.
- NTDD: comportamiento definido en como se envian mensajes los objetos.

ATDD (Acceptance Test Driven Development):

Definición: es una práctica de diseño de software que obtiene el producto a partir de las pruebas de aceptación. Se basa en TDD, pero en vez de escribir pruebas unitarias (que escriben y usan los programadores), escriben pruebas de aceptación de usuario, en conjunto con ellos.

Idea: tomar cada requerimiento, en forma de user story, construir varias pruebas de aceptación de usuario y a partir de ellas construir las pruebas automáticas de aceptación, para luego escribir el código.

User story: requerimientos simples y no representan el cómo, sino solamente quién necesita qué y por qué.

Ventajas:

- A los clientes no les interesa el código, sino que el producto satisfaga sus necesidades con la mejor calidad posible, ATDD permite saber cuando una necesidad se ha satisfecho: sólo hay que ver que la prueba de aceptación basada en esa user story funque.
- Ayuda a eliminar posibles ambigüedades entre qué quiere el cliente y qué entiende el desarrollador.

La premisa fundamental es que TDD se concibió para diseñar y probar código, pero los clientes no están interesados en el código, sino en que el sistema satisfaga sus necesidades con la mejor calidad posible. ATDD permite conocer mejor cuándo se ha satisfecho un requerimiento, tanto para el desarrollador como para el cliente: simplemente hay que preguntar si todas las pruebas de aceptación de las user story están funcionando.

BDD: TDD mejorado

Idea:

- ❖ En vez de hacer pruebas de clases y métodos, lo más lógico es probar porciones de comportamiento esperados del sistema.
- ❖ En vez de pensar en términos de pruebas, deberíamos pensar en términos de especificaciones o comportamiento.
- ❖ Escribir las pruebas desde el punto de vista del consumidor y no del producto.

Prácticas:

- ❖ Cambiar todos los métodos test por should o must.
- ❖ Escribir una clase de prueba por requerimiento y un método por aserción.

- Escribir los nombres de los métodos para que se puedan leer como oraciones.
- Hablar en el lenguaje del cliente ayuda mantener la alta abstracción, sin caer en detalles de implementación.

Críticas:

- Un cambio de nombre a TDD, cuidando un poco el lenguaje.
- TDD bien hecho.

STDD: ejemplos como pruebas y pruebas como ejemplos

Idea: usar ejemplos como partes de las especificaciones

- los desarrolladores construyen su código y escriben pruebas unitarias, para las cuales deben basarse en ejemplos de entradas y salidas.
- los testes desarrollan casos de pruebas, que contienen a su vez ejemplos.
- a veces los desarrolladores y los testers le solicitan a los analistas que les den ejemplos concretos para aclarar ideas y otras veces son los mismos analistas los que proveen escenarios, que no son más que requerimientos en la forma de ejemplos.

Ventajas de los requerimientos con ejemplos:

- sirven como herramienta de comunicación
- evita las largas descripciones y reglas de prosa, propensas a interpretaciones diversas.
- son más sencillos de acordar con los clientes
- sirven como pruebas de aceptación

Desventajas:

- A veces ocurre que al plantear los requerimientos como ejemplos se pierde la visión global del proyecto.
- No todo requerimiento puede llevarse a un ejemplo. Ejemplo: una aplicación que debe generar un número al azar, los ejemplos que podamos escribir nunca van a servir como prueba de aceptación.
- No resulta fácil escribir ejemplos para pruebas de estrés o de desempeño.

Roles del enfoque tradicional:

- analistas: hace de traductor entre los clientes y los desarrolladores y testers
- tester: valida el producto contra las especificaciones. Recibe especificaciones, elabora casos de prueba en base a escenario y ejecuta casos de prueba.
- desarrollador: diseñador y constructor del producto
- cliente o usuario: solo habla con el analista.

Roles en el enfoque STDD:

- Analista: deja de ser un traductor para ser un facilitador de intercambio de conocimiento

- Cliente: tiene un rol más activo, como autor principal de pruebas de aceptación con ejemplos, asistido por el resto de los interesados

Taller:

- intercambiar ideas entre los participantes
- se reduce la brecha entre lo que entiende uno y otro participante

El foco de diseño orientado a objetos: NDD

Ley de Demeter: pretende que los objetos revelen lo mínimo posible de su estado interno, dando a conocer sólo su comportamiento.

Idea: el comportamiento de los objetos debe estar definido en términos de cómo envía mensajes a otros objetos, además de los resultados devueltos en respuesta a mensajes recibidos.

Propósito: obtener un buen diseño orientado a objetos y una separación clara de responsabilidades, sobre la base de:

- mejorar el código en términos de dominio
- preservar el encapsulamiento
- reducir dependencias
- clarificar las interacciones entre clases

Pretende definir de antemano qué mensajes emitirá el objeto sometido a prueba cuando reciba un mensaje en particular.

Utiliza Mock Objects: porque nos interesa la comunicación entre objetos y no su estado o implementación.

Ventajas:

- Interfaces bien angostas.

Desventajas:

- Sensible a refactorizaciones, ya que al renombrar o sacar métodos, las pruebas dejan de funcionar ya que los Mock Objects trabajan buscando métodos por nombre.

Pruebas de interacción y de TDD

Pruebas e interfaz de usuarios

Problemas a la hora de hacer pruebas:

- la interfaz de usuario es cambiante, por lo que las pruebas se desactualizan con mucha frecuencia.
- Suelen ser muy lentas de ejecutar
- Hay cuestiones que las puede probar satisfactoriamente un usuario, por ejemplo, la ubicación de los botones, los colores, la consistencia de la aplicación en su totalidad.

Pero por qué es importante hacerlas:

- El valor de la aplicación se muestra a través de la interfaz de usuario.
- La mayoría de los usuarios reales sólo consideran que la aplicación les sirve cuando la ven a través de su interfaz de usuario.

Cómo se prueban: casi todas las herramientas permiten grabar una prueba realizada a mano para después ejecutarla de forma repetitiva.

Limitaciones de las pruebas de interacción:

Problemas que suelen presentar las pruebas grabadas:

- Sensibilidad al comportamiento: los cambios en el comportamiento generan cambios importantes en la interfaz de usuario, que hacen que las pruebas de interacción dejen de funcionar.
- Sensibilidad a la interfaz: aún pequeños cambios a la interfaz provocan que las pruebas dejen de correr y deban ser cambiadas.
- Sensibilidad a los datos: si se cambian los datos con los que corren las pruebas, los resultados arrojados por las mismas van a cambiar, lo que hace que haya que generar datos especiales para probar.
- Sensibilidad al contexto: las pruebas pueden ser sensibles a cambios en el dispositivo externos a la aplicación.

Consejos:

- Tratar de no probar al mismo tiempo la lógica del negocio con la lógica de la interacción.
- Evitar hacer estas pruebas automatizadas si la lógica del negocio es muy cambiante.
- Volver a generar las pruebas cada tanto.

Tipos de pruebas y automatización

Que automatizar