Design Principles Behind Smalltalk

Dan Ingalls

Principios generales:

- *Dominio personal:* si un sistema debe servir al espíritu creativo, debe ser completamente comprensible para un solo individuo.

- *Buen diseño:* un sistema debe construirse con un conjunto mínimo de piezas inalterables; esas partes deben ser lo más generales posible; y todas las partes del sistema deben mantenerse en un marco uniforme.

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

- *Objetos:* un lenguaje informático debe respaldar el concepto de "objeto" y proporcionar un medio uniforme para referirse a los objetos en su universo.

- *Gestión de almacenamiento:* para estar verdaderamente "orientado a objetos", un sistema informático debe proporcionar una gestión de almacenamiento automática.

- *Mensajes:* la computación debe verse como una capacidad intrínseca de los objetos que se pueden invocar de manera uniforme mediante el envío de mensajes.

- *Metáfora uniforme:* un lenguaje debe diseñarse en torno a una metáfora poderosa que se pueda aplicar de manera uniforme en todas las áreas.

Organización

Una metáfora uniforme proporciona un marco en el que se pueden construir sistemas complejos. Varios principios organizacionales relacionados contribuyen a la gestión exitosa de la complejidad. Para empezar:

- *Modularidad:* ningún componente de un sistema complejo debe depender de los detalles internos de cualquier otro componente.

- *Clasificación:* un lenguaje debe proporcionar un medio para clasificar objetos similares y para agregar nuevas clases de objetos en pie de igualdad con las clases del núcleo del sistema.

- *Polimorfismo:* un programa debe especificar solo el comportamiento de los objetos, no su representación.

- *Factorización:* cada componente independiente de un sistema aparecería en un solo lugar.

- *Apalancamiento:* cuando un sistema está bien factorizado, tanto los usuarios como los implementadores disponen de un gran apalancamiento.

- *Máquina virtual:* una especificación de máquina virtual establece un marco para la aplicación de la tecnología.

Interfaz de usuario

Una interfaz de usuario es simplemente un lenguaje en el que la mayor parte de la comunicación es visual.

- *Principio reactivo:* cada componente accesible para el usuario debe poder presentarse de manera significativa para su observación y manipulación.

- *Sistema operativo:* un sistema operativo es una colección de cosas que no encajan en un idioma. No debería haber uno.

- *Selección natural:* Los lenguajes y sistemas que tienen un diseño sólido persistirán, para ser suplantados solo por otros mejores.

Unit Testing Guidelines

Petroware SA

1. Mantenga las pruebas unitarias pequeñas y rápidas.
Idealmente, todo el conjunto de pruebas debe ejecutarse antes de que cada código se registre. Mantener las pruebas rápidas reduce el tiempo de respuesta del desarrollo.

2. Las pruebas unitarias deben ser completamente automatizadas y no interactivas.
El conjunto de pruebas normalmente se ejecuta de forma regular y debe estar completamente automatizado para que sea útil. Si los resultados requieren una inspección manual, las pruebas no son pruebas unitarias adecuadas.

3. Haga que las pruebas unitarias sean simples de ejecutar.
Configure el entorno de desarrollo para que las pruebas individuales y los conjuntos de pruebas se puedan ejecutar con un solo comando o con un solo clic.

4. Medir las pruebas.
Aplique el análisis de cobertura a las ejecuciones de prueba para que sea posible leer la cobertura de ejecución exacta e investigar qué partes del código se ejecutan y no.

5. Solucione las pruebas fallidas de inmediato
Cada desarrollador debe ser responsable de asegurarse de que una nueva prueba se ejecute correctamente al registrarse y que todas las pruebas existentes se ejecuten correctamente al registrar el código.

Si una prueba falla como parte de una ejecución de prueba regular, todo el equipo debe abandonar lo que está haciendo actualmente y asegurarse de que se solucione el problema.

6. Mantenga las pruebas a nivel de unidad

Las pruebas unitarias se tratan de clases de prueba . Debe haber una clase de prueba por clase ordinaria y el comportamiento de la clase debe probarse de forma aislada. Evite la tentación de probar un flujo de trabajo completo utilizando un marco de pruebas unitarias, ya que dichas pruebas son lentas y difíciles de mantener. Las pruebas de flujo de trabajo pueden tener su lugar, pero no son pruebas unitarias y deben configurarse y ejecutarse de forma independiente.

7. Comience de manera simple

Una simple prueba es infinitamente mejor que ninguna prueba. Una clase de prueba simple establecerá el marco de prueba de la clase objetivo, verificará la presencia y corrección tanto del entorno de construcción, el entorno de prueba unitaria, el entorno de ejecución y la herramienta de análisis de cobertura, y probará que la clase objetivo es parte de el conjunto y que se pueda acceder a él.

8. Mantenga las pruebas independientes

Para garantizar la solidez de las pruebas y simplificar el mantenimiento, las pruebas nunca deben depender de otras pruebas ni del orden en que se ejecutan las pruebas.

9. Mantenga las pruebas cerca de la clase que se está evaluando

Si la clase a probar es Foo, la clase de prueba debe llamarse FooTest (no TestFoo) y mantenerse en el mismo paquete (directorio) que Foo . Mantener las clases de prueba en árboles de directorios separados hace que sea más difícil acceder a ellas y mantenerlas.
Asegúrese de que el entorno de compilación esté configurado para que las clases de prueba no lleguen a las bibliotecas o ejecutables de producción.

10. Nombra las pruebas correctamente

Asegúrese de que cada método de prueba pruebe una característica distinta de la clase que se está probando y nombre los métodos de prueba en consecuencia. La convención de nomenclatura típica es test[what] como testSaveAs() , testAddListener() , testDeleteProperty() etc.

11. Prueba la API pública

Las pruebas unitarias se pueden definir como clases de prueba a través de su API pública . Algunas herramientas de prueba permiten probar el contenido privado de una clase, pero esto debe evitarse ya que hace que la prueba sea más detallada y mucho más difícil de mantener. Si hay contenido privado que parece necesitar pruebas explícitas, considere refactorizarlo en métodos públicos en clases de utilidad. Pero haga esto para mejorar el diseño general, no para ayudar en las pruebas.

12. Piensa en una caja negra

Actúe como un consumidor de clase de terceros y pruebe si la clase cumple con sus requisitos. Y tratar de destrozarlo.

13. Piensa en una caja blanca

Después de todo, el programador de prueba también escribió la clase que se está probando, y se debe hacer un esfuerzo adicional para probar la lógica más compleja.

14. Prueba también los casos triviales

A veces se recomienda probar todos los casos no triviales y omitir los métodos triviales como setters y getters simples. Sin embargo, hay varias razones por las que también se deben probar los casos triviales:

- Trivial es difícil de definir. Puede significar diferentes cosas para diferentes personas.
- Desde una perspectiva de caja negra, no hay forma de saber qué parte del código es trivial .
- Los casos triviales también pueden contener errores, a menudo como resultado de operaciones de copiar y pegar.

Por lo tanto, la recomendación es probar todo . Después de todo, los casos triviales son fáciles de probar.

15. Centrarse primero en la cobertura de ejecución

Distinguir entre cobertura de ejecución y cobertura de prueba real . El objetivo inicial de una prueba debe ser garantizar una alta cobertura de ejecución. Esto asegurará que el código se ejecute realmente en algunos parámetros de entrada. Cuando esto está en su lugar, la cobertura de la prueba debe mejorarse. Tenga en cuenta que la cobertura de prueba real no se puede medir fácilmente (y de todos modos siempre está cerca del 0%).

16. Cubrir casos límite

Asegúrese de que los casos límite de los parámetros estén cubiertos. Para números, negativos de prueba, 0, positivo, más pequeño, más grande, NaN, infinito, etc. Para cadenas, pruebe cadena vacía, cadena de un solo carácter, cadena que no sea ASCII, cadenas de varios MB, etc. Para colecciones, pruebe vacía, uno, primero, último, etc. Para las fechas, pruebe el 1 de enero, el 29 de febrero, el 31 de diciembre, etc. La clase que se está probando sugerirá los casos límite en cada caso específico. El punto es asegurarse de que la mayor cantidad posible de estos se prueben correctamente, ya que estos casos son los principales candidatos para errores.

17. Proporcione un generador aleatorio

Cuando se cubren los casos límite, una forma simple de mejorar aún más la cobertura de la prueba es generar parámetros aleatorios para que las pruebas se puedan ejecutar con diferentes entradas cada vez.

Para lograr esto, proporcione una clase de utilidad simple que genere valores aleatorios de los tipos base como dobles, enteros, cadenas, fechas, etc. El generador debe producir valores del dominio completo de cada tipo.

Si las pruebas son rápidas, considere ejecutarlas dentro de bucles para cubrir tantas combinaciones de entrada como sea posible.

18. Prueba cada característica una vez

Cuando se está en modo de prueba, a veces es tentador afirmar "todo" en cada prueba. Esto debe evitarse ya que dificulta el mantenimiento. Pruebe exactamente la característica indicada por el nombre del método de prueba.

En cuanto al código ordinario, el objetivo es mantener la cantidad de código de prueba lo más baja posible.

19. Usa afirmaciones explícitas

Prefiere siempre afirmarEquals(a, b) a afirmarVerdadero(a == b) (y de la misma manera) ya que el primero dará información más útil de qué es exactamente lo que está mal si la prueba falla. Esto es particularmente importante en combinación con parámetros de valores aleatorios como se describe anteriormente cuando los valores de entrada no se conocen de antemano.

20. Proporcionar pruebas negativas

Las pruebas negativas hacen un mal uso intencional del código y verifican la solidez y el manejo adecuado de errores.

21. Diseñe código pensando en las pruebas

Escribir y mantener pruebas unitarias es costoso, y minimizar la API pública y reducir la complejidad ciclomática en el código son formas de reducir este costo y hacer que el código de prueba de alta cobertura sea más rápido de escribir y más fácil de mantener.

Algunas sugerencias:

- Haga que los miembros de la clase sean inmutables al establecer el estado en el momento de la construcción. Esto reduce la necesidad de métodos setter.
- Restringir el uso de herencia excesiva y métodos públicos virtuales.
- Reduzca la API pública utilizando clases amigas (C++), alcance interno (C#) y alcance del paquete (Java).
- Evite las ramificaciones innecesarias.
- Mantenga la menor cantidad de código posible dentro de las ramas.
- Haga un uso intensivo de excepciones y afirmaciones para validar argumentos en API públicas y privadas, respectivamente.
- Restringir el uso de métodos de conveniencia. Desde una perspectiva de caja negra, todos los métodos deben probarse igualmente bien.

22. No te conectes a recursos externos predefinidos

Las pruebas unitarias deben escribirse sin un conocimiento explícito del contexto del entorno en el que se ejecutan para que puedan ejecutarse en cualquier lugar y en cualquier momento. Para proporcionar los recursos necesarios para una prueba, estos recursos deben estar disponibles por la propia prueba.

Considere, por ejemplo, una clase para analizar archivos de cierto tipo. En lugar de seleccionar un archivo de muestra de una ubicación predefinida, coloque el contenido del archivo dentro de la prueba, escríbalo en un archivo temporal en el proceso de configuración de la prueba y elimine el archivo cuando finalice la prueba.

23. Conozca el costo de las pruebas

No escribir pruebas unitarias es costoso, pero escribir pruebas unitarias también lo es. Existe una compensación entre los dos, y en términos de cobertura de ejecución, el estándar típico de la industria es de alrededor del 80 %.

Las áreas típicas en las que es difícil obtener una cobertura completa de la ejecución son el manejo de errores y excepciones al tratar con recursos externos. Es bastante posible simular un desglose de la base de datos en medio de una transacción, pero podría resultar demasiado costoso en comparación con las revisiones exhaustivas del código, que es el enfoque alternativo.

24. Prioriza las pruebas
La prueba unitaria es un proceso ascendente típico, y si no hay suficientes recursos para probar todas las partes de un sistema, la prioridad debe colocarse primero en los niveles inferiores.

25. Prepare el código de prueba para fallas
Siempre prepárese para la falla de la prueba para que la falla de una sola prueba no detenga la ejecución de todo el conjunto de pruebas.

26. Escribir pruebas para reproducir errores
Cuando se informe un error, escriba una prueba para reproducir el error (es decir, una prueba fallida) y utilice esta prueba como criterio de éxito al corregir el código.

27. Mantenlo simple
Las pruebas unitarias deben ser simples para ser efectivas, no deben contener una complejidad integral por sí mismas. Un olor seguro es si la prueba unitaria está duplicando parte de la lógica en el código que se está probando, o si parece que el código de prueba en sí mismo necesita una prueba unitaria.

28. Conoce las limitaciones
¡Las pruebas unitarias nunca pueden probar la corrección del código!
Una prueba fallida puede indicar que el código contiene errores, pero una prueba exitosa no prueba nada en absoluto.

El dispositivo más útil de las pruebas unitarias es la verificación y documentación de los requisitos a bajo nivel y las pruebas de regresión: verificar que las invariantes del código permanezcan estables durante la evolución y la refactorización del código.

En consecuencia, las pruebas unitarias nunca pueden reemplazar un diseño inicial adecuado y un proceso de desarrollo sólido. Las pruebas unitarias deben utilizarse como un valioso complemento de las metodologías de desarrollo establecidas.

Y quizás lo más importante: el uso de pruebas unitarias obliga a los desarrolladores a pensar en sus diseños que, en general, mejoran la calidad del código y las API.

¿What Is the Point of Test-Driven Development?

Freeman & Pryce (Capítulo 1 del libro "Growing object-oriented software, guided by tests", Addison-Wesley - 2011)

Desarrollo de software como proceso de aprendizaje

Los desarrolladores usualmente no entienden completamente la tecnología que están usando, tienen que aprender cómo funciona sobre la marcha.

El feedback es una herramienta fundamental

Un equipo necesita repetir ciclos de cada actividad y en cada ciclo se agregan nuevas características y se obtiene un feedback sobre la cantidad y calidad del trabajo hecho.

- *En cada loop el desarrollo es incremental e iterativo*

- *El desarrollo incremental:* consiste en construir un sistema, característica a característica (pasito a pasito), en vez de construir todos los componentes por separado e integrarlos al final.

- *El desarrollo iterativo:* refina la implementación de las diferentes características en base al feedback hasta que sea suficientemente bueno

Prácticas que apoyan el cambio

Para sistemas de cualquier tamaño interesante, las pruebas manuales frecuentes no son prácticas, por lo que debemos automatizar las pruebas tanto como podamos.
puede reducir los costos de construir, implementar y modificar versiones del sistema.

- Necesitamos pruebas constantes para detectar errores de regresión, para que podamos agregar nuevas funciones sin romper los existentes

- Debemos mantener el código lo más simple posible, para que sea más fácil de entender y modificar.

TDD en pocas palabras

La regla de oro de TDD:

- Nunca escriba una nueva funcionalidad sin un test que falle.

Refactorización. Piense localmente, actúe localmente

Refactorizar significa cambiar la estructura interna de un cuerpo de código existente sin cambiar su comportamiento. El punto es mejorar el código para que sea una mejor representación de las funciones que implementa, haciéndolo más fácil de mantener.

La refactorización es una “microtécnica” que se basa en encontrar mejoras a pequeña escala.

La fotografia mas grande
seguimos el ciclo de prueba/implementación/refactorización a nivel de unidad para desarrollar la función

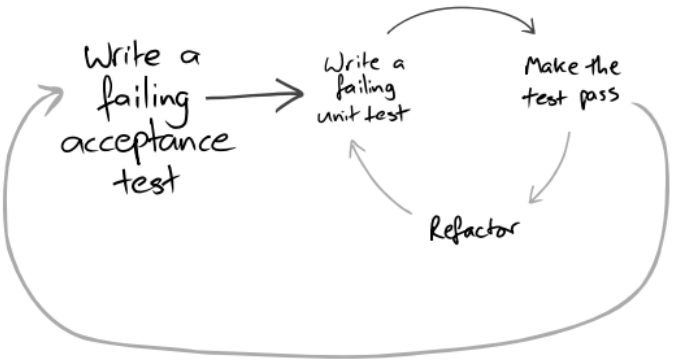


Figure 1.2 *Inner and outer feedback loops in TDD*

Testing end-to-end
Una prueba de extremo a extremo interactúa con el sistema solo desde el exterior: a través de su interfaz de usuario, enviando mensajes como si fueran sistemas de terceros, invocando sus servicios web, analizando informes, etc.

Niveles de testeo
- *Aceptación:* ¿Funciona todo el sistema?
- *Integración:* ¿Nuestro código funciona contra el código que no podemos cambiar?
- *Unidad:* ¿Nuestros objetos hacen lo correcto, son convenientes para trabajar con ellos?

Calidad externa e interna
- *calidad externa:* satisface las necesidades de sus clientes y usuarios (es funcional, confiable, disponible, receptivo, etc.).
- *calidad interna:* satisface las necesidades de sus desarrolladores y administradores (es fácil de entender, fácil de cambiar, etc.).

Acoplamiento y cohesión
Son métricas que describen qué tan fácil es cambiar el comportamiento de algún código
- Elementos están acoplados si un cambio en uno fuerza un cambio en otro
- Un elemento tiene cohesión si sus responsabilidades tienen relaciones lógicas entre sí

8 Principles of Better Unit Testing

Dror Helper

¿Qué hace que un test unitario sea bueno?
Los test unitarios son tests cortos, rápidos y automatizados que aseguran que una parte específica de su programa funcione. Prueban la funcionalidad específica de un método o clase que tiene una condición clara de aprobación/rechazo.

Un test unitario "bueno" sigue estas reglas:

- El test solo falla cuando se introduce un nuevo error en el sistema o cambian los requisitos.
- Cuando el test falla, es fácil entender la razón de la falla.

Para escribir buenos test unitarios se deben seguir las siguientes pautas:

1. *Sepa lo que está probando:*

- Una prueba escrita sin un objetivo claro en mente es fácil de detectar. Este tipo de prueba es larga, difícil de entender y carece de enfoque, por lo general, evalúa más de una cosa.

2. *Las pruebas unitarias deben ser autosuficientes:*

- Una buena prueba unitaria debe estar aislada. Evite dependencias como la configuración del entorno. Una sola prueba no debe depender de la ejecución de otras pruebas antes de ella, ni debe verse afectada por el orden de ejecución de otras pruebas.

3. *Las pruebas deben ser deterministas:*

- La peor prueba es la que pasa parte del tiempo. Una prueba debe pasar todo el tiempo o fallar hasta que se solucione. Tener una prueba unitaria que pasa parte del tiempo es equivalente a no tener ninguna prueba. Además hay que evitar pruebas con entradas aleatorias.

4. *convenciones de nomenclatura:*

- Para saber por qué falló una prueba, debemos poder entenderla de un vistazo. Hay que utilizar buenos nombres que describan qué es lo que hace la prueba

5. *Repitase:*

- Está permitido tener código duplicado en las diferentes pruebas unitarias, ya que esto mejora su legibilidad.

6. *Resultados de la prueba, no implementación:*

- Solo pruebe métodos privados si tiene una muy buena razón para hacerlo. La refactorización trivial puede causar errores de complicación y fallas en las pruebas.

7. *Evite la especificación excesiva:*
- No hacer pruebas muy específicas

8. *Use un marco de aislamiento:*
- Las dependencias dificultan la capacidad de escribir pruebas unitarias. Cuando tales dependencias necesitan una configuración compleja para que ejecute la prueba automatizada, hay que utilizar mock objects para evitar pruebas frágiles que se rompan.

The Art of Enbugging

Andy Hunt & Dave Thomas

Enbuggin:
Es que los programadores escriban los bugs por su cuenta, no intencionalmente

Los bugs no aparecen espontáneamente en nuestro código, y hay forma de evitarlos desde el comienzo. Una gran forma es realizando una separación de intereses, que consiste en diseñar el código con las clases y módulos bien definidos, con responsabilidades aisladas y nombres entendibles.

La meta fundamental es escribir *Shy Code:* código que no revela mucho por sí mismo y no habla con otros más de lo necesario

Tell Don't Ask:
Hay que decirle a los objetos que hacer y no preguntar por su estado interno

Una buena forma de respetar tell don't ask y shy code es separando los metodos en:
- *Command:* cambia el estado del objeto y devuelve un valor útil
- Query: solo entrega información sobre el estado del objeto pero no puede cambiarlo

Ley de Demeter:
Esta idea sugiere que un objeto solo debe llamar:
- A si mismo
- Cualquier parámetro que se le pasó en el método
- Cualquier objeto que crea
- Cualquier objeto retenido directamente

La desventaja de este método es:
- Ineficiencia versus el acoplamiento de clase superior

El acoplamiento de clase superior es simplemente inaceptable, ya que aumenta las probabilidades de que cualquier cambio que realice rompa algo en otra parte. Pero para aquellas ocasiones en las que la velocidad es primordial y el alto acoplamiento es aceptable, acoplalo al palo, no seas tímido.

GetterEradicator

Martin Fowler

Para Martin Fowler el objetivo de la encapsulación no es realmente ocultar los datos, sino ocultar las decisiones de diseño, particularmente en áreas donde esas decisiones pueden tener que cambiar

Cuando pensas en encapsulamiento, es mejor preguntarse a uno mismo “¿qué pieza de variabilidad está ocultando y por qué?" en lugar de "estoy exponiendo datos"

Se habla de eliminar todos los getters cuanto sea posible ya que estos violan el encapsulamiento, una razón más válida para eliminar los getters es que de esta forma se nos empuja al lugar correcto

Kent Beck:
No siempre se pueden eliminar los getters pero hay que tener cuidado
cuando un método tiene más de una solicitud de ese estilo y pensar si hay una forma de que solo haya un getter y no muchos.

Otras buena señal de advertencia son las clases anémicas, que sólo tienen atributos, getters y setters pero no poseen ningún comportamiento. Si ves esto debes sospechar.

Craig Larman llama ”experto en información” a poner el comportamiento en la misma clase que los datos. Una buena regla general es que las cosas que cambian juntas deben estar juntas. Los datos y el comportamiento que los usa a menudo cambian juntos

Replace Conditional With Polymorphism

SourceMaking

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

Solución: crear subclases que coincidan con las ramas condicionales. En ellos cree un método compartido y mueva el código de la rama correspondiente del condicional a el. Luego reemplace el condicional con la llamada al método relevante

Por qué refactorizar:
Esta técnica de refactorización puede ayudar si su código contiene operadores que realizan varias tareas que varían según:

- Clase del objeto o interfaz que implementa
- Valor del campo de un objeto
- Resultado de llamar a uno de los métodos de un objeto

Beneficios:

- Esta técnica se adhiere al principio Tell don't ask.
- Elimina código duplicado
- Respeta Open/Closed

¿Para qué sirve un modelo?

Martin Fowler

Es importante reconocer que un diagrama UML involucra un costo y un modelo UML no es importante para el cliente. Osea que el valor del modelo está altamente ligado a su impacto en el software. Si el modelo mejora el código, entonces tiene valor. Pero el modelo no tiene valor por sí mismo.

Hay que saber que tanta información poner en el modelo, y cuán relevante es para su posterior interpretación

Debe servir para resaltar las partes importantes del sistema pero no ignorar los detalles

En la opinión de martin fowler un modelo esquelético es mejor que un modelo de cuerpo entero, ya que cuando estoy tratando de entender un modelo debo darme cuenta de donde empezar.

Un modelo de cuerpo esquelético es más fácil de mantener que uno de cuerpo entero. Los detalles importantes cambian con menos frecuencia. Y si alguien quiere más detalles debe explorar el código.

Pruebas de software

Carlos Fontela

Contexto:
se desarrollar un producto de calidad, trabajando mejor, para que el control de la calidad, sea casis siempre exitoso utilizando el aseguramiento de la calidad (QA)

Verificación y validación:
- *Verificación:* tiene que ver con controlar que hayamos construido el producto tal como pretendíamos construirlo
- *Validación:* en cambio, controla que hayamos construido el producto que el cliente quiera

Funcionalidades y atributos de calidad:
Pruebas funcionales: son pruebas centradas en lo que el programa debe hacer

Pruebas de atributos de calidad:
probar características del sistema que no son funcionales.

Alcance de las pruebas:
Las pruebas de verificación son pruebas que los propios desarrolladores ejecutan para ver que están logrando que el programa funcione como ellos pretenden, se categorizan en unitarias o de integración.

Alcance de las pruebas de verificación:
- *pruebas unitarias:* verifican pequeñas porciones de código.

- *Pruebas de integración:* prueban varias porciones del código.
- A veces no es sencillo hacer una prueba unitaria porque estamos probando código que necesita de otros objetos, para esto se utilizan los mock objects, que simulan comportamientos

- *Prueba de caja negra:* es cuando la ejecutamos sin ver el código que se está probando.

- *Prueba de caja blanca:* es cuando vemos el código que estamos estamos probando, el debugging es una técnica de caja blanca

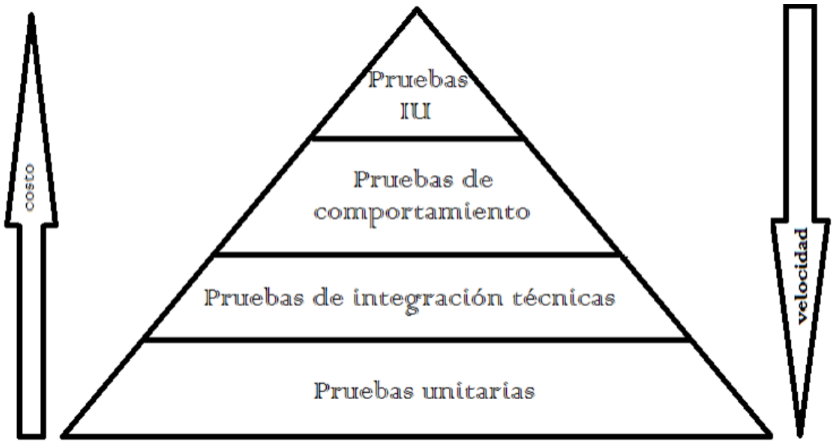
- *Prueba de escritorio:* el programador da valores a variables y recorre mentalemnte para ver si se comporta como espera.

- *Revisión de código de a dos programadores:*

Alcance de las pruebas de validación:
- Pruebas de aceptación de usuarios (UAT). Pruebas diseñadas por o con usuarios, ejecutadas en un entorno parecido al que va a utilizar el usuario.

- Cuando se pone el sistema a disposición de usuarios reales para que lo prueben.
- *Pruebas alfa:* si se realizan en un entorno controlado por el equipo de desarrollo.
- *Pruebas beta:* si dejamos que el usuario lo pruebe en su entorno.
- Cuando queremos probar solamente que la lógica de la app es correcta sin la interfaz de usuario usamos pruebas de comportamiento.
- Pruebas en producción, es entregar el producto sin probar y esperar a ver que errores surgen o qué problemas encuentran los usuarios.

La pirámide de pruebas y sus razones - Es un tema económico - Las unitarias son más rápidas y menos costosas, entonces hay más.



Roles del desarrollo ante las pruebas:

- Visiones tradicionales: Debe haber programadores y testers. Los testers son quienes ejecutan las pruebas y velan porque el producto llegue sin errores a los clientes y usuarios mientras que los programadores deben intentar entregar un codigo sin errores para luego ser chequeado

- *La visión ágil:* No hay separación de roles, todo el equipo trabaja en conjunto en pos de la calidad. O pueden separarse pero esto depende de como quiera trabajar el equipo

- *Pruebas automatizadas:* ¿Quién las desarrolla?
  Unitarias: programador
  Integración: programador
  Comportamiento: programador y tester
  UAT: usuarios o analistas de negocios y en su defecto testers

Pruebas y desarrollo:

- Hoy en día las pruebas son continuas a lo largo de todo el desarrollo
- Cada vez que agregó nuevas características a un programa puedo romper las viejas, esto se llama regresión, para evitarlas están las pruebas de regresión, que simplemente es ejecutar todas las pruebas regularmente

Ventajas de la automatización:

- Nos independizamos del factor humano
- Es más fácil repetir las pruebas
- Sirven como herramienta de comunicación

TDD: Desarrollo guiado por pruebas

Pruebas orientadas al cliente:

- TDD tenía un problema ya que minimizó la importancia de las pruebas de aceptación así que surgieron otras prácticas que sí contemplan eso:
    - BDD (Behaviour Driven Development): surgió como una práctica para hacer bien el TDD y fue mutando a una manera de especificar el comportamiento esperado mediante escenarios que se puedan automatizar como pruebas de aceptación

    - STDD (Story Test Driven Development): es una practica que pretende construir el software basándose en ir haciendo pasar las pruebas de aceptación

    - ATDD (Acceptance test driven development): similar a la anterior, construye el producto en base a las pruebas de aceptación con menos énfasis en la automatización y más en el proceso en sí

    - SBE (Specification By Example): presentada originalmente como una práctica para mejorar la comunicación entre distintos roles de un proyecto, se convirtió en una práctica colaborativa de construcción basada en especificaciones mediante ejemplos que sirven como pruebas de aceptación

Integración y entregas continuas:

- *Integración continua(CI):* consiste en realizar la compilación, construcción y pruebas del producto en forma sucesiva y automática

- *Entrega continua (CD):* pretende que el código siempre esté en condiciones de ser desplegado en el ambiente productivo. Las pruebas no se ejecutan constantemente

- Despliegue continuo: Codigo anda todo el tiempo y todos los pequeños cambios se prueban

Cobertura:

- Es el porcentaje de casos contemplados en las pruebas
- Es razonable un 80% de cobertura
- Tener en cuenta que hay zonas del código más críticas donde todo debe ser probado al 100%
- Entender que tener una cobertura alta
- no quiere decir que sea una cobertura buena

Continuous Integration

Martin Fowler

La esencia de esto radica en la simple práctica de que todos los miembros del equipo se integren con frecuencia, generalmente a diario, en un repositorio de código fuente controlado.

La integración continua es una práctica que no requiere de herramientas particulares para implementar, pero es útil utilizar un servidor de integración continua, el más conocido de este tipo es *CruiseControl*, es una herramienta de código abierto.

Creación de una característica con integración continua:
Bajo la línea principal (el main) y lo edito localmente en mi computadora para integrar las características nuevas. Luego hago pasar las pruebas automatizadas y subo todo al main.

Pueden generarse problemas si dos personas actualizan el main al mismo tiempo, y es responsabilidad del último en realizar la integración arreglarlo.

Mantener un repositorio de origen único:

- Asegúrese de obtener un sistema de administración de código fuente decente.

- Todo debería estar en el repositorio. Scripts de prueba, archivos de propiedades, esquema de base de datos, etc. No tenes que tener algo que usas solo vos para una característica. Todo debe estar compartido.

- Los sistemas de control de versiones nos permiten crear múltiples ramas para manejar diferentes flujos de desarrollo. Prácticamente todo el mundo debería trabajar fuera de la línea principal.

Automatice la construcción:
Se tiene que poder hacer funcionar y subir todo con solo un comando.

Haga que su compilación se autoevalúe:

- Pruebas automatizadas que verifiquen gran parte de la base del código, si una prueba falla, la compilación debe fallar.
- Utilizar herramientas XUnit para autoevaluar el código.
- Las pruebas imperfectas que se ejecutan con frecuencia son mejores que las pruebas perfectas que nunca se escribieron

Todos se comprometen con la línea principal todos los días:

- Debe haber comunicación entre los miembros del equipo
- Cuanto más a menudo se comprometan, menos lugares tendrá para buscar errores de conflicto y más rápidamente solucionará los conflictos.

Cada compromiso debe construir la línea principal en una máquina de integración:

- Con confirmaciones diarias, un equipo obtiene compilaciones probadas con frecuencia

- Un servidor de integración continua verifica automáticamente lo que se subió y comprueba y notifica el resultado de la compilación

- Muchas organizaciones realizan compilaciones regulares en un horario cronometrado, como todas las noches. Esto no es lo mismo que una integración continua y no es suficiente

Repare las compilaciones rotas de inmediato:

- Kent Beck: “Nadie tiene una tarea de mayor prioridad que arreglar el build”

- La forma más rápida de corregir es regresar el main a una versión anterior y luego depurar en busca del error en nuestra versión

Mantenga la construcción rápida:

- Una compilación de diez minutos está perfectamente dentro de lo razonable

- Canalización de implementación, también conocida como compilación por etapas, consiste en que hay varias compilaciones realizadas en secuencia

- El commit build, es la que debe hacerse rápidamente

- Lo que más tarda al hacer la compilación son las cosas que tengan que ver con recursos externos por eso conviene separar la compilación, por un lado realizando las pruebas unitarias que son rápidas y concentrándose en los problemas que esas pueden encontrar y a la vez haciendo las pruebas que llevan más tiempo. Así optimizamos el tiempo.

Probar en un clon del entorno de producción:

- El objetivo de las pruebas es eliminar, en condiciones controladas, cualquier problema que el sistema pueda tener en producción.

Facilite que cualquiera tenga el ultimo ejecutable:

(no hay nada que agregar aca la verdad)

Todos pueden ver lo que está pasando.

- La integración continua tiene que ver con la comunicación, por lo que desea asegurarse de que todos puedan ver fácilmente el estado del sistema y los cambios que se le han realizado.
- Se debe poder ver si alguien está compilando en el momento.

Automatice la implementación:
Para realizar la integración continua, necesita varios entornos, uno para ejecutar pruebas de confirmación, uno o más para ejecutar pruebas secundarias. Dado que está moviendo ejecutables entre estos entornos varias veces al día, querrá hacerlo automáticamente.

Beneficios de la integración continua:

- En general creo que el mayor y más amplio beneficio de la integración continua es la reducción del riesgo.

- El problema de la integración diferida (hacer todo por separado y después juntarlo), es que es muy difícil predecir cuánto tardará, y no se ve que tan avanzado está el proyecto

- La integración continua no elimina los errores pero los hace más fáciles de encontrar

- Síndrome de las ventanas rotas: las personas tienen menos ganas y energía de encontrar errores si hay demasiados

Introducción a la integración continua:

- Automatizar la compilación

- Introduce pruebas automatizadas en tu compilación

- Intente acelerar el commit build y mantenerlo siempre por debajo de la regla de los 10 minutos en cuanto empiece a volverse lento por dependencias externas.

- Si inicias un nuevo proyecto, comience la integración continua desde el principio

Estado del arte y tendencias en Test-Driven Development

Carlos Fontela

TDD como práctica metodológica:
Test Driven Development es una práctica iterativa presentada por Kent Beck y Ward Cunningham como parte de Extreme Programming

Incluye tres sub prácticas:

- *Automatización:* permite independizarse del factor humano y facilita la repetición de las pruebas

- *Test - First:* Escribir las pruebas antes que el código minimiza el condicionamiento del autor por lo ya construido y da más confianza al desarrollador

- *Refactorización posterior*: facilita el mantenimiento de un buen diseño

- *Regla de oro:* Nunca escribas nuevas funcionalidad sin una prueba que falle antes

La aparición de los frameworks y el corrimiento de TDD a UTDD:
Frameworks de pruebas automatizadas: SUnit (para smalltalk) y JUnit (java) - xUnit : genérico

Las mayores ventajas de UTDD son:
- Las pruebas indican con menor ambigüedad lo que las clases y métodos deben hacer
- Las pruebas incluyen más casos de pruebas negativas

Desventajas:
- Tiende a basar todo el desarrollo en la programación de pequeñas unidades, sin una visión de conjunto.
- No permite probar interfaces de usuario ni comportamiento esperado por el cliente
- Es una práctica centrada en la programación que no sirve para especialistas del negocio ni testers
- Los cambios de diseño medianos y grandes suelen exigir cambios en las pruebas unitarias

Los primeros intentos de pruebas de integración:
Como siempre que hablamos de integración, el mayor problema es como probar interacciones con módulos aún no implementados. La idea más simple es construir módulos ficticios o stubs.

Hay distintos tipos:
- Dummy Object: son aquellos que deben generarse para probar una funcionalidad, pero que no se usan en la prueba.

- Test Stub: son lo que reemplazan a objetos reales del sistema, generalmente para generar entradas de datos o impulsar funcionalidades del objeto que está siendo probado.

- Test Spy: se usan para verificar los mensajes que envía el objeto que se está probando, una vez corrida la prueba.

- Mock Object: son objetos que reemplazan a objetos reales del sistema para observar los mensajes enviados a otros objetos desde el receptor.

- Fake Object: son objetos que reemplazan a otro objeto del sistema con una implementación alternativa.

Con los objetos ficticios se busca disminuir las dependencias y que las pruebas de integración se mantengan como unitarias.

La visión de las pruebas de cliente de XP:
Kent Beck planteó que debía haber una presencia de un representante del cliente junto al equipo de desarrollo y este debía participar en la creación de pruebas funcionales

El punto de vista de los requerimientos y del comportamiento:
ATDD: TDD ampliado:

- Acceptance Test Driven Development es una práctica que obtiene el producto a través de las pruebas de aceptación. La lógica es la de TDD pero con pruebas de aceptación escritas junto con los usuarios.

- La idea es tomar cada requerimiento en la forma de una user story (caso de uso) y a partir de ellas construir las pruebas automáticas de aceptación.

- Lo importante de las user stories es que son requerimientos simples y no representan el cómo, sino solamente quien necesita que y por que.

- Evita requerimientos que el cliente da por sobreentendidos y el gold-plating de los desarrolladores (agregar funcionalidades que el cliente nunca pensó).

BDD: TDD mejorado

- Behaviour Driven Development, impulsado por Dan North, e influenciado por Eric Evans y su Domain Driven Design (DDD), consiste en pensar en términos de especificaciones o comportamiento, y no tanto en términos de prueba. Así se puede validar más fácilmente con clientes y especialistas de negocio. Escribiendo las pruebas desde el punto de vista del consumidor y no del productor.

- Las primeras herramientas para realizar BDD pusieron mucho énfasis en cambios de nombres. Suprimiendo los métodos test y cambiandolos por should o must.

- También se asoció a escribir pruebas por requerimiento (user story o casos de uso) - Las clases y los métodos se deben escribir con lenguaje de negocio, haciendo que los nombres se puedan leer como oraciones.

Desventajas y críticas:

- Se dice que BDD es solo un cambio de nombre a TDD y no aporta mucho
- Se dice que BDD es solo TDD bien hecho

BDD vs ATDD:
La principal diferencia es que mientras que en BDD escribimos casos de uso de forma tradicional, ATDD enfatiza que las pruebas de aceptación también las escriban los clientes.

Las coincidencias radican en sus objetivos generales

- Mejorar las especificaciones
- Mejorar la comunicación entre los distintos perfiles
- Mejorar la visibilidad de la satisfacción de requerimientos y del avance
- Disminuir el gold-plating

- Usar un lenguaje único más cerca del consumidor
- Simplificar las refactorizaciones

La crítica principal viene de un teórico de la POO, Bertrand Meyer. Según él son los contratos los queden dirigir el diseño y las pruebas.

La respuesta habitual de Ward Cunningham es que si bien las especificaciones abstractas son más generales, las pruebas concretas son más fáciles de comprender

STDD: ejemplos como pruebas y pruebas como ejemplos:
Story Test Driven Development, consiste en usar ejemplos como parte de las especificaciones, y que los mismos sirvan para probar la aplicación.

Ventajas:
- Los ejemplos sirven como herramienta de comunicación
- Se expresan por extensión, en vez de con largas descripciones
- Son más sencillos de acordar con los clientes, al ser más concretos
- Sirven como pruebas de aceptación

STDD a recibido otras denominaciones
- Brian Marick lo llama “especificación con ejemplos” o “pruebas de cara al negocio”
- Gerald Weinberg y Donald Gause lo llaman “requerimientos de caja negra”
- Gojko Adzic lo llama “especificaciones con ejemplos”, “requerimientos guiados por pruebas” o “pruebas de aceptación ágiles”

Limitaciones de la práctica de especificar con ejemplos:
Epics:
- Término que se usa para referirse a iteraciones que se hacen para lograr un conocimiento más global.
- No todo requerimiento puede llevarse a ejemplos.

Formatos de las especificaciones
- *tablas:* son buenas para representar entradas y salidas calculables, pero malas para representar flujos de tareas. No es facil pasarlo a pruebas
- *texto libre*: todos pueden entenderlo, pero no es facil pasarlo a pruebas
- *diagramas*: buenos para flujos, malos para entradas y salidas. Difícil pasar a pruebas
- *especificaciones de código:* fácil pasar a pruebas, son malas para la comunicaciones

El foco del diseño orientado a objetos NDD:
El problema del comportamiento:
- *Ley de Demeter:* pretende que los objetos revelen lo mínimo posible de su estado interno, dando a conocer solamente su comportamientos

- Los frameworks tradicionales verifican el comportamiento de los objetos acarreando dos problemas

- No siempre el comportamiento puede evaluarse por los cambios de estado o por valores que devuelve

- Para chequear el estado a veces necesitamos métodos de consulta que la clase no tiene, nos vemos obligados a cambiar el código solo para las pruebas violando el encapsulamiento

Una propuesta de solución: NDD

- Steve Freeman plantea que el comportamiento de los objetos debería estar definido en términos de cómo enviar mensajes entre objetos

- NDD propone implementar los servicios como objetos ficticios o algo asi

Recomendaciones al usar NDD

- Generar objetos ficticios solamente de las clases que podemos cambiar
- Solo generar Mock Objects a partir de interfaces
- Evitar el uso de métodos de consulta
- Si se usan muchos Mock Objects en una prueba quiere decir que el objeto receptor tiene muchas responsabilidades

Pruebas de interacción y TDD:
Pruebas de interfaz de usuario:

- lentas y normalmente es necesaria que las haga un ser humano

- se pueden automatizar grabando una secuencia que la computadora repite

Limitaciones de las pruebas de interacción:

- “El problema de la prueba frágil”

- Sensibilidad al comportamiento: los cambios de comportamiento provocan cambios importantes en la interfaz de usuario

- Sensibilidad a la interfaz: aun cambios pequeños a la interfaz de usuario suelen provocar que las pruebas dejen de correr y deban ser cambiadas

- Sensibilidad a los datos: cuando hay cambios en los datos que se usan para correr la app, los resultados que arroje van a cambiar

- Sensibilidad al contexto: Las pruebas suelen ser sensibles a cambios en dispositivos externos a la aplicación

Tipos de pruebas y automatización:
Qué automatizar:

- Gerard Meszaros propone la siguiente clasificación, Pruebas de cliente, pruebas de componentes, pruebas de unidad, pruebas de usabilidad, pruebas exploratorias, pruebas de propiedades

Formas de automatización:

- hay dos maneras de interactuar con la aplicación a probar
- Mediante la interfaz de usuario
- Mediante una interfaz programática (API)

TDD y enfoques metodológicos:

- Enfoque botton-up (de menor a mayor dificultad) y top-down (de mayor a menor dificultad)

- Fowler propone hacer lo que él llama “diseño de afuera hacia adentro”. En esta metodología se parte de la interfaz de usuario y se pasa a la capa media y así sucesivamente. El problema es que así necesitas muchos mock objects

Using Java Reflection

Glen McCluskey

Reflection es una característica del lenguaje Java. Permite que un programa en ejecución examine sobre sí mismo y manipule propiedades internas del programa

La reflexión de java es útil porque admite la recuperación dinámica de información sobre clases y estructuras de datos por nombre, y permite su manipulación dentro de un programa java en ejecución[1]

[1] Caspian 2C2023 - Fiesta