ArtículoS . UncleBob .

TheThreeRulesOfTdd [agregar niño]

Las tres leyes de TDD.

A lo largo de los años he llegado a describir el desarrollo basado en pruebas en términos de tres reglas simples. Son:

1. No está permitido escribir ningún código de producción a menos que sea para aprobar un pase de prueba de la unidad.
2. No se le permite escribir más de una prueba unitaria que sea suficiente para fallar; y las fallas de compilación son fallas.
3. No está permitido escribir ningún código de producción más de lo que es suficiente para pasar la prueba de falla de la unidad.

Debe comenzar por escribir una prueba unitaria para la funcionalidad que tiene la intención de escribir. Pero según la regla 2, no se puede escribir mucho de esa prueba unitaria. Tan pronto como el código de prueba de la unidad no compila o falla una afirmación, debe detenerse y escribir el código de producción. Pero según la regla 3, solo puede escribir el código de producción que hace que la prueba compile o pase, y no más.

Si piensas en esto, te darás cuenta de que simplemente no puedes escribir mucho código sin compilar y ejecutar algo. De hecho, este es realmente el punto. En todo lo que hacemos, ya sea escribiendo pruebas, escribiendo código de producción o refactorizando, mantenemos el sistema ejecutándose en todo momento. El tiempo entre las pruebas en ejecución es del orden de segundos o minutos. Incluso 10 minutos es demasiado largo.

Demasiado ver esto en funcionamiento, eche un vistazo aEl juego de bolos Kata .

Ahora la mayoría de los programadores, cuando escuchan por primera vez sobre esta técnica, piensan: "¡ *Esto es estúpido!* " " *Me va a retrasar, es una pérdida de tiempo y esfuerzo, me evitará pensar, me impedirá diseñar, simplemente interrumpirá mi flujo* " . Sin embargo, piense en lo que sucedería si entrara en una habitación llena de gente que trabaja de esta manera. Elige cualquier persona al azar en cualquier momento al azar. Hace un minuto, todo su código funcionó.

Permítanme repetir eso: ¡ *Hace un minuto todo su código funcionó!* Y no importa a quién elija, y no importa cuando elige. **¡Hace un minuto todo su código funcionó!**

Si todo tu código funciona a cada minuto, ¿con qué frecuencia usarás un depurador? Responde, no muy a menudo. Es más fácil simplemente presionar ^ Z un montón de veces para que el código vuelva a funcionar, y luego tratar de escribir los últimos minutos valiosos nuevamente. Y si no está depurando mucho, ¿cuánto tiempo ahorrará? ¿Cuánto tiempo pasas depurando ahora? ¿Cuánto tiempo pasas solucionando errores una vez que los depura? ¿Qué pasaría si pudieras disminuir ese tiempo en una fracción significativa?

Pero el beneficio va más allá de eso. Si trabajas de esta manera, cada hora estás realizando varias pruebas. Todos los días docenas de pruebas. Todos los meses cientos de pruebas. En el transcurso de un año, escribirás miles de pruebas. ¡Puede mantener todas estas pruebas y ejecutarlas cuando lo desee! ¿Cuándo los ejecutarías? ¡Todo el tiempo! ¡Cada vez que hiciste algún tipo de cambio!

¿Por qué no limpiamos el código que sabemos que es complicado? Tememos que lo rompamos. Pero si tenemos las pruebas, podemos estar razonablemente seguros de que el código no está roto, o que vamos a detectar la rotura de inmediato. Si tenemos las pruebas, nos damos miedo de hacer cambios. Si vemos un código desordenado o una estructura sucia, podemos limpiarlo sin temor. Debido a las pruebas, el código vuelve a ser maleable. Debido a las pruebas, el software vuelve a ser suave.

Pero los beneficios van más allá de eso. Si desea saber cómo llamar a cierta API, existe una prueba que lo hace. Si desea saber cómo crear un determinado objeto, existe una prueba que lo hace. Todo lo que quieras saber sobre el sistema existente, hay una prueba que lo demuestra. Las pruebas son como pequeños documentos de diseño, pequeños ejemplos de codificación, que describen cómo funciona el sistema y cómo usarlo.

¿Alguna vez ha integrado una biblioteca de terceros en su proyecto? Tienes un gran manual lleno de buena documentación. Al final había un apéndice delgado de ejemplos. ¿Cuál de los dos leíste? Los ejemplos por supuesto! ¡Eso es lo que las pruebas de la unidad son! Son la parte más útil de la documentación. Son los ejemplos vivientes de cómo usar el código. Son documentos de diseño que son horriblemente detallados, absolutamente no ambiguos, tan formales que se ejecutan y que no pueden deshacerse de la sincronización con el código de producción.

Pero los beneficios van más allá de eso. Si alguna vez ha intentado agregar pruebas unitarias a un sistema que ya estaba funcionando, probablemente descubrió que no fue muy divertido. Es probable que descubriera que tenía que cambiar partes del diseño del sistema o hacer trampa en las pruebas; porque el sistema para el que intentabas escribir pruebas no estaba diseñado para ser comprobado. Por ejemplo, le gustaría probar alguna función 'f'. Sin embargo, 'f' llama a otra función que elimina un registro de la base de datos. En su prueba, no desea que se elimine el registro, pero no tiene forma de detenerlo. El sistema no fue diseñado para ser probado.

¡Cuando sigas las tres reglas de TDD, *todo tu código será comprobable por definición!* Y otra palabra para "comprobable" es "desacoplada". Para probar un módulo de forma aislada, debe desacoplarlo. Entonces TDD te obliga a desacoplar módulos. De hecho, si sigues las tres reglas, te encontrarás haciendo mucho más desacoplamiento del que estarás acostumbrado. Esto te obliga a crear diseños mejores, menos acoplados.

Dados todos estos beneficios, estas pequeñas reglas estúpidas de TDD podrían no ser tan estúpidas. En realidad, podrían ser algo fundamental, algo profundo. De hecho, había sido programador durante casi treinta años antes de que me presentaran a TDD. No pensé que alguien pudiera enseñarme una práctica de programación de bajo nivel que marcaría la diferencia. Treinta años es *mucho*de experiencia después de todo. Pero cuando comencé a usar TDD, me quedé estupefacto ante la efectividad de la técnica. Yo también estaba enganchado. Ya no puedo imaginar escribir un gran lote de código esperando que funcione. Ya no puedo tolerar arrancar un conjunto de módulos, con la esperanza de volverlos a ensamblar y hacer que funcionen todos para el próximo viernes. Cada decisión que tomo mientras programo está motivada por la necesidad básica de volver a ejecutar en un minuto a partir de ahora.

▼ *Viernes, 7 de octubre de 2005 03:37:42, Johan Nilsson, la reparación de errores* [Expandir todo](#) | [Desplegar todo](#)

Esto podría estar un poco fuera de tema, pero ¿qué le parece agregar 'No tiene permitido arreglar ningún código sin tener un caso de prueba reproducible (unidad)'? OK, no se aplica al desarrollo inicial, pero los errores aparecen incluso en sistemas desarrollados usando TDD

(aunque son raros cuando se aplican correctamente). ¿O es este TDBF - 'Prueba de corrección de errores' ;-)

▼ *Vie, 7 de octubre de 2005, 08:16:14, Craig Demyanovich, Corrección de errores* Expandir todo | Desplegar todo

Para mi proyecto actual, todavía en desarrollo, creamos pruebas fallidas para corregir errores. Usar las ideas que presenta el tío Bob nos permite soltarlo a menudo. Invariablemente, hay algunas cosas menores que no hicimos bien, o hay algunos casos de historias que ni el cliente ni los desarrolladores consideraron. Tomamos esa retroalimentación, creamos una prueba de falla para cada elemento y lo hacemos pasar. Demostramos inmediatamente la corrección al cliente, que sabe que el problema no estará presente en la próxima versión. Entonces, Johan, no veo ninguna razón por la cual esta técnica no pueda aplicarse durante toda la vida de la aplicación.

▼ *Viernes, 7 de octubre de 2005 19:47:51, Justicia, ¿Qué hay de la IU?* Expandir todo | Desplegar todo

Entonces, ¿qué haces para el código de la interfaz de usuario (por ejemplo, en una aplicación web)? Sé que un poco de eso puede estar desacoplado, pero no todo ...

▼ *Sáb, 8 de octubre de 2005, 04:51:40,* Expandir todo | Desplegar todo

▼ *Sáb, 8 de octubre de 2005 a las 05:05:30, Sagy Rozman, TDD para el código de la interfaz de usuario* Expandir todo | Desplegar todo

Hay muchos ejemplos de aplicaciones web escritas usando TDD.
Puede consultar los diferentes grupos de discusión de yahoo relacionados con este tema.
Mejor aún: descargue la fuente de Fitnesse, es una aplicación web. No lo he comprobado, pero apuesto a que puedes encontrar algunos buenos ejemplos de TDD para los módulos de UI, ya que fue (¿sobre todo?) Escrito por el propio Robert Martin.

- *Fue escrito principalmente por Micah Martin, pero ayudé un poco. == UB.*

▼ *Sáb, 8 de octubre de 2005 a las 05:10:08, Sebastian Kübeck,* Expandir todo | Desplegar todo

Hay más cosas que no pueden probarse en una unidad ...

http://www.artima.com/weblogs/viewpost.jsp?thread=126923

Creo que las reglas deben reducirse principalmente a la lógica del dominio para que sea factible.

Todas esas cosas pueden (y deben) probarse, el único problema es si usted cuenta esas pruebas como pruebas unitarias o no. - Plumas de Michael

▼ *Sáb, 8 de octubre de 2005, 11:46:17, Sebastian Kübeck,* Expandir todo | Desplegar todo

> Todas esas cosas se pueden (y se deben) probar,
> el único problema es si cuentas esas pruebas como pruebas unitarias o no. - Michael Feathers

Las reglas son explícitamente sobre pruebas unitarias.
Creo que el truco consiste en mantener el código que no se puede probar con la unidad lo más pequeño posible
e introducir pruebas automatizadas y reproducibles para el código que, por ejemplo, se comunica directamente
con la base de datos.
Es necesario un reemplazo (por ejemplo, en el almacén de datos de memoria) para el código que no
es comprobable por la unidad para mantener las pruebas de la unidad de manera rápida e independiente del mundo exterior.

▼ *Lunes, 10 de octubre de 2005, 14:33:05, tío Bob. Código que no se puede probar por unidad.* Expandir todo | Desplegar todo

Tengo otra regla. No es parte de los tres, pero es parte de mi repertorio personal. *Regla 4: No existe el código no comprobable.* . ¿Es esta regla verdadera? No estoy seguro Sin embargo, lo trato como una actitud. Me niego a creer que, dada la creatividad e imaginación apropiadas, haya algún código que no pueda ser probado. De hecho, me niego a creer que, dada la creatividad e imaginación apropiadas, haya un código que no pueda ser probado de manera *rentable* .

Aceptar que hay ciertos tipos de código que son "imposibles" de probar es una pendiente resbaladiza. Es una excusa para abandonar la creatividad y someterse a la canción de la sirena del programa que susurra que no tienes tiempo, que no tienes tiempo, que no tienes tiempo.

Si tiene un código que cree que no se puede probar, le sugiero que:
1. Detente, y encuentra una manera de probarlo.
2. Encuentra una forma de cambiarlo para que se pueda probar.

Martes, 11 de octubre de 2005, 09:44:13, Sebastian Kuebeck, Expandir todo | Desplegar todo

Gran punto! Haría que la Regla 4 realmente fuera la Regla 1 ¡ya que es la más importante! Estoy absolutamente de acuerdo con tu opinión de que todo debería probarse automáticamente.
El punto es el tiempo que tarda una prueba en ejecutarse.
Mantengo pruebas lentas (por ejemplo, llamadas a procedimientos almacenados que necesitan intercambiar datos
entre varias tablas) en un directorio diferente al de las rápidas que dirigen mi desarrollo, así que no estoy bloqueado por una infraestructura lenta durante el desarrollo (TDD es una palabra diferente para la obsesión de la flecha verde ;-)).
También llamo a los lentos con frecuencia, especialmente antes o durante el descanso.
Tal vez estoy equivocado, pero pensé que solo los rápidos eran llamados pruebas unitarias y la lenta integración o cualquier otra prueba, eso es todo.

Mar, 11 de octubre de 2005 19:43:05, Keith Gregory, ¿Qué pasa si la siguiente prueba lógica requiere más de 10 minutos de codificación? Expandir todo | Desplegar todo

Me gusta la idea de TDD, y creo firmemente que "si es difícil de probar, será difícil de usar". Sin embargo, siento que debo perderme una visión más profunda, porque parece haber muchos casos en los que la implementación de un objeto no crece linealmente con las pruebas.

Por ejemplo: este fin de semana traté de implementar una Lista de saltos usando las reglas de TDD de este artículo. Omitir lista es un reemplazo para árboles binarios, que usa un generador de números aleatorios para distribuir entradas (ver http://en.wikipedia.org/wiki/Skip_list). La API externa es simple, generalmente siguiendo otras colecciones de Java. Entonces, una serie razonable de pruebas para esta API parece ser: 1) Construir, 2) Agregar objeto, 3) Verificar tamaño, 4) Recuperar objeto.

Sin embargo, después de esta secuencia, existe una gran brecha de implementación entre las pruebas 1 y 2: toda la administración interna de las entradas. Por supuesto, puedo diferir esa implementación, tal vez usando una lista enlazada de un solo nivel con inserción para llegar hasta la prueba 4, pero eso realmente no me lleva en la dirección de una lista de omisiones. La mayoría de la implementación "interesante" ocurre en una clase interna y en un par de métodos privados en la clase externa, y no quiero exponerlos solo con el propósito de probar.

Un punto ligeramente diferente es la verificación de que mi implementación realmente construye una serie de listas vinculadas que tienen la distribución correcta. Probar este hecho es fundamental para saber que tengo un objeto que almacenará / recuperará en tiempo O (log2N) versus tiempo O (N). Mi única respuesta dentro de los límites de "probar la interfaz" es agregar un método de distribución a la interfaz. Quizás sea útil para otros fines, pero existe para probar.

▼ *Jue, 13 de octubre de 2005 13:42:10, George Dinwiddie, código inestable* Expandir todo | Desplegar todo

"I have another rule. It's not part of the three, but it is part of my personal repertoire. Rule 4: There is no such thing as untestable code."

Hmmm... I think I've seen untestable code. I would change Rule 4 to Commandment 1: *Thou shalt not write untestable code*. It's not that code that does certain things is untestable, but that people write it in a way that's untestable.

Back in the early 1980s, when I was working in hardware development, there was a mantra in the trade magazines, "Design for testability." This included things such as having or being able to set a known starting state for a circuit, and having access to sufficient internal nodes to determine proper operation. As hardware development moved from discrete circuits to custom integrated circuits, these issues came to the fore because a technician was more limited in the transformations he could make to allow testing. The same issues face the software development community.

▼ *Thu, 13 Oct 2005 19:20:53, Moz, So how do I test UI code?* Expand All | Collapse All

> If you have code that you believe cannot be tested, then I suggest that you:
> 1. Stop, and find a way to test it.
> 2. Find a way to change it so that it's testable.

A huge amount of what I do is UI or interface code. Making sure that (eg) drag and drop works properly is interesting and tricky to test, but it's nothing compared to some of the more intense UI code. Call this method, watch it alter a spreadsheet... some kind of test to make sure that all the values are correct is necessary, but a bit outside the scope of most testing systems.

Currently we're using TestComplete[?] and screenshots, but to say that that sucks is an understatement. The user enabling font aliasing is a slightly different error to all the values in a spreadsheet being wrong, but the test result is the same...

So I wonder how people automatically test things like this?

- *By decoupling! You create the bevavior you want OUTSIDE the UI, and then use the UI to render it. This often means that you don't make use of convenient facilities in the UI. Instead you create presentation models that behave like the UI, but that do not contain any UI code. And finally you wire those presentation models to the UI using very simple glue code. == UB*

▼ *Wed, 19 Oct 2005 14:36:37, Dave Bouwman, UI Testing...* Expand All | Collapse All

We've got a little bit of a problem with UI testing, and heres a quick run down of why.
We develop customizations for the ESRI ArcGIS system (www.esri.com for more on that). Since GIS is very very heavy on UI interactions, and we're building on someone elses framework and/or application, we're at a loss for how to test it. We've looked at test complete and other "snapshot" type systems, and they're not worth implementing for the scale of projects we do (month or two at a time). We try to decouple as much as possible, but since we rely on ESRI's underlying objects, and interface objects, there are many cases where we can not decouple.

Ideas?

Dave

▼ *Mon, 21 Nov 2005 03:14:31, Carlos C Tapang, What if the next logical test requires more than 10 minutes of coding?* Expand All | Collapse All

Has there been a response to Keith Gregory's question above? If there has, I am missing it.

I suppose the answer is to write tests for the inner class also?

Tue, 22 Nov 2005 14:19:20, John D. Mitchell, Skip lists and TDD Expand All | Collapse All

Keith,

In terms of testing, your existing 'unit' tests are basically just API-level functional tests. If you want to make meta-data (e.g., statistics) about the data structure part of your API, that's up to you. However, given your aims for the module, it sounds like you really need some 'white-box'/internal/private tests.

In terms of TDD, I don't see how you could have possibly implemented that inner class and those private methods by following e.g., the 'rules' of TDD. I.e., how did you have failing tests each step of the way that lead you to all of that (internal) functionality? It sounds like you must have taken some awfully larger steps in a premeditated direction in your coding.

Tue, 22 Nov 2005 17:24:52, Uncle Bob, What if the next logical test requires more than 10 minutes of coding? Expand All | Collapse All

> Carlos C Tapang asked: *Has there been a response to Keith Gregory's question above? If there has, I am missing it.*

First, I *would* start with a simple linear list implementation. The reason I would do that is that it would prove that my tests are correct. Any tests that pass for a linear list, shoudl pass for a skip list. Eventually, in order to get the Skip List implementation, I'd have to write some tests that a linear list would fail, and that only a skip list would pass. I would choose very tiny incremental test cases so that I did not have to implement the whole Skip List in one great leap. Choosing these tiny incremental tests is part of the *art* of TDD.

Finally, I think you should lose the attitude: "Just for the tests." I don't mind exposing a variable, function, or inner class if it helps me write the tests. I don't find that I have to do it too often; but I don't mind doing it. Also the extra 'distribution' function doesn't bother me. Again, I don't find that I have to do things like that very often, but the tests are so important that I certainly will expose functions, or add utilities, if it supports the tests.

Mon, 28 Nov 2005 17:52:47, Keith Gregory, What if the next logical test requires more than 10 minutes of coding? Expand All | Collapse All

|Uncle Bob writes: Finally, I think you should lose the attitude: "Just for the tests." I don't mind exposing a variable, function, or inner class if it helps me write the tests.

Perhaps. My concern, however, is that breaking encapsulation is a slippery slope. Today it's just for the tests, tomorrow it's because another class becomes simpler when it has access to the internals of the first. I've met a bunch of smart programmers who slid down that slope, and am trying not to join their ranks.

To that end, I came up with three alternative approaches:

First is to create a "statistics" object that's updated by the internal methods of SkipList[?] and read by the tests. Not quite a mock object, but same idea. The primary risk here is collecting the wrong statistics. I also think that putting a lot of work into the mock disrupts the flow of small increments.

Second, implement "in vitro" and move the tested code into the final class. You end up with "published" tests for the API alone, but development proceeds in small increments.

Third, extract the core of the list into its own class, and test that while you're developing it.

Probably the cleanest approach, but it seems that unnecessary *decoupling* is a smell -- it goes against "do the simplest thing that can possibly work."

|Choosing these tiny incremental tests is part of the *art* of TDD.

That's true, and I'd like to see some exploration of that art in a realistic setting. The Bowling Game is a nice way to demonstrate TDD on a micro scale, but let's face it, it works because it has a limited story: "score a bowling game after all the balls have been rolled." (btw, how do you avoid an IndexOutOfBounds[?] in testPerfectGame()?)

▼ *Fri, 2 Dec 2005 15:03:44, John Cowan, Bah* Expand All | Collapse All

Type the first character of a test, compile it; it fails, naturally. Type the next character of a test, compile it; it fails, naturally. Repeat. We keep rules 1, 2, and 3, but we never deliver anything.

▼ *Fri, 2 Dec 2005 16:15:56, Bill Krueger, Managability of 1000's of test cases* Expand All | Collapse All

With the pro's come the con's. In this case, there's having to deal with the potentially overwhelming number (1000's) of test cases. For the reasons they you propose they have worth for examples and documentation, the numbers seem to negate that.

That said, I do like your thoughts.

▼ *Sat, 3 Dec 2005 19:21:37, Uncle Bob, Bah: We never deliver anything.* Expand All | Collapse All

> John Cowan said:*Type the first character of a test, compile it; it fails, naturally. Type the next character of a test, compile it; it fails, naturally. Repeat. We keep rules 1, 2, and 3, but we never deliver anything.*

John, Perhaps you are taking this too literally. There are many examples available for how to apply these laws. Consider, for example, TheBowlingGameKata.

▼ *Sat, 3 Dec 2005 19:28:56, Uncle Bob, 1000s of test cases.* Expand All | Collapse All

Yes, there can be thousands of test cases. The more production code you have, the more test cases you will have. Fortunately, the structure of the tests mimics the structure of the production code. For every class, there is often a corresponding test-class. This test-class describes how the class works, how to call it, and what to expect from it. Personally, I keep the test classes in the same package as the production code class. So when I look in my IDE I see the class, and the test-class right next to each other. That's *very* nice.

▼ *Mon, 5 Dec 2005 11:38:17, Matt Segvich, Re: Managability of 1000's of test cases* Expand All | Collapse All

> Bill Krueger said:*With the pro's come the con's. In this case, there's having to deal with the potentially overwhelming number (1000's) of test cases.*

From my experience, I've never had to deal with all the test cases at once, rather just the ones that are impacted by the changes. Which for me is typically a handful. The only time I've seen others struggle is when they put off testing instead of doing it in-step with the changes. The longer you put them off, the more tests you'll need to update. Also remember, to find the ones that you've broke you just run the tests. For me the tests allow me to sleep at night. In fact, more often than not I find (due to bugs) I have too few tests.
