Engineer

Desarrollo guiado por pruebas

Escribir la prueba antes del código, una pequeña cada vez

Qué es

El desarrollo guiado por pruebas es una forma de programar en la que la prueba de un comportamiento se escribe antes que el código que lo proporciona, una pequeña prueba cada vez. Kent Beck, que describe su propia participación como un redescubrimiento, lo denomina un flujo de trabajo de programación. En 2023, tras comprobar que los programadores no coincidían sobre el significado del término, expuso el flujo en una breve publicación titulada Canon TDD y utilizó la abreviatura TDD en casi todo el texto.

El flujo de trabajo tiene cinco pasos:

- Enumerar los casos que debe atender el cambio.
- Elegir exactamente un caso de la lista y convertirlo en una prueba concreta que se pueda ejecutar.
- Hacer que esa prueba pase sin romper ninguna anterior y añadir a la lista todos los casos nuevos que salgan a la luz durante el trabajo.
- Si el diseño de la implementación puede mejorar, refactorizarlo ahora; este paso es opcional.
- Volver al paso 2 hasta que no quede nada en la lista.

  La lista es un análisis del comportamiento, no un documento de diseño: primero el caso básico, después las variantes y luego las formas en que el cambio no debe romper lo que ya funciona. La prueba escrita en el paso 2 es una prueba automatizada real, con preparación, una invocación y aserciones, y falla cuando se ejecuta por primera vez porque el comportamiento que describe aún no se ha escrito. Beck llama al paso 3 pasar de rojo a verde, de una prueba que falla a una que pasa.

También resulta útil saber qué queda fuera de este flujo de trabajo. Una crítica frecuente al TDD es que obliga a escribir todas las pruebas antes de escribir código, y Beck responde directamente: eso no es TDD, y criticarlo es criticar otra cosa. Tampoco exige rigidez en la dirección contraria. Un equipo que trabaja de otra manera y obtiene buenos resultados puede seguir haciéndolo; según Beck, no se concede ninguna estrella por seguir los pasos al pie de la letra.

Utilice el concepto en una decisión

Aplique «¿Qué es el desarrollo guiado por pruebas?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. Un equipo que lleva una idea del boceto a un producto al que sus clientes pueden acceder, sin pasarla entre tres proveedores.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Ingresos multiplicados por 10 con mapas GIS — 10x crecimiento de ingresos impulsado desde el producto cartográfico

Ingresos multiplicados por 10 con mapas GIS Póngase en contacto →

Por qué importa

Beck explica para qué sirve el flujo de trabajo. Un programador que cambia el comportamiento de un sistema debería terminar con un sistema en el que el cambio conserve el comportamiento existente y cumpla los requisitos nuevos, el código esté preparado para el siguiente cambio y tanto el programador como sus compañeros confíen en esos tres aspectos. Como cada prueba se escribe cuando el cambio aún es pequeño y reciente, la red de seguridad crece al mismo ritmo que el código, en lugar de prometerse para más adelante.

La segunda razón es el diseño. Beck distingue dos tipos de decisiones de diseño: cómo se invoca un comportamiento y cómo lo implementa el sistema. Escribir primero la prueba obliga a decidir la interfaz, porque la prueba tiene que llamar a algo y decir qué devuelve. Las decisiones de implementación esperan al paso de refactorización, cuando las protegen pruebas que pasan. Una prueba difícil de escribir avisa pronto de que la interfaz será difícil de utilizar.

La tercera razón es la doble comprobación. Una persona calcula el valor esperado de una prueba a partir del requisito, y el código llega a él por otra vía. Si ambos coinciden, dos caminos independientes han alcanzado la misma respuesta, y Beck afirma que buena parte del valor del TDD como comprobación procede de esa doble verificación.

También hay una orientación sobre cuándo parar. Beck aconseja seguir probando y programando hasta que el temor sobre cómo se comporta el código se convierta en aburrimiento. La práctica es mucho más antigua que la publicación. Beck sitúa su redescubrimiento entre 1994, cuando escribió SUnit para un cliente de consultoría, y octubre de 1995, cuando mostró el TDD a Ward Cunningham en la conferencia OOPSLA de Austin. Su artículo Simple Smalltalk Testing describe un marco de pruebas sencillo para Smalltalk, y la documentación de unittest de Python sigue remitiendo a él como el artículo original de Beck sobre el patrón que comparte unittest.

Cómo se aplica en la práctica

Considere una comprobación de contenido que deba señalar palabras de moda prohibidas en los textos de un sitio web. El trabajo empieza con la lista de pruebas, escrita como simples notas:

```
- se señala una palabra prohibida
- se señala una palabra prohibida con una letra mayúscula
```

El paso 2 convierte exactamente un elemento en una prueba. El primer elemento ocupa ese lugar por una razón: proporciona a la comprobación una palabra prohibida conocida y verifica que se informe de ella. La prueba falla hasta que la comprobación puede detectarla. Se guarda en un archivo llamado `test_wording.py`, que pytest recoge porque el nombre coincide con `test_*.py`:

```
from wording import find_banned


def test_reports_a_banned_word():
    assert find_banned("we leverage it", {"leverage"}) == ["leverage"]
```

Ejecutar `pytest -q` falla de inmediato porque no existe ninguna función `find_banned` que importar. Ese fallo es lo que se busca: se ha visto fallar la prueba, por lo que un resultado satisfactorio posterior tendrá significado. De momento basta con el código más sencillo que la haga pasar, en `wording.py`:

```
def find_banned(text, banned):
    return [word for word in text.split() if word in banned]
```

La prueba pasa. El siguiente elemento se convierte en la siguiente prueba:

```
def test_reports_a_capitalised_banned_word():
    assert find_banned("Leverage it", {"leverage"}) == ["Leverage"]
```

Falla, y como pytest muestra los valores intermedios de una aserción, la salida presenta una lista vacía donde se esperaba `["Leverage"]`. Comparar `word.lower()` en lugar de `word` hace que pase. Mientras escribe ese cambio, el programador observa que `text.split()` deja la puntuación unida a las palabras, por lo que «leverage,» con una coma pasaría inadvertida. Eso se añade a la lista, en lugar de al código, y se convierte en la tercera prueba:

```
def test_reports_a_banned_word_before_a_comma():
    assert find_banned("Use it, leverage, win", {"leverage"}) == ["leverage"]
```

Falla, y una línea más hace que pasen las tres:

```
def find_banned(text, banned):
    words = [word.strip(".,;:!?") for word in text.split()]
    return [word for word in words if word.lower() in banned]
```

El paso 4 es opcional y aquí hay poco que mejorar: dos líneas, cada una con un cometido. Cada elemento de la lista ya tiene una prueba que pasa, así que el trabajo ha terminado. Cada valor esperado de las pruebas se escribió a partir del elemento de la lista, sin copiarlo nunca de una ejecución.

El orden de esa lista fue deliberado, y el registro del propio sitio muestra por qué. Un proyecto que escribió pruebas para sus propios comprobadores, realizado en agosto de 2026, descubrió que sus comprobaciones personalizadas, entre ellas las de ortografía, voz y coherencia de cifras, llevaban meses dando resultados satisfactorios y que no se puede distinguir un comprobador que solo ha visto entradas correctas de otro que no lee nada. La solución fue introducir el defecto que cada comprobador debe encontrar: al corrector ortográfico se le proporciona una palabra escrita deliberadamente mal y la prueba falla si el resultado no señala errores; al comprobador de voz se le proporciona prosa en primera persona y debe rechazarla; y al de palabras de moda, una palabra prohibida. Resultó que dos comprobadores leían un conjunto de archivos más

reducido de lo que afirmaba su documentación. La página no describe TDD, pero su enseñanza es la que el TDD incorpora desde el primer paso: una prueba solo tiene valor una vez que se la ha visto fallar.

Dónde surgen los problemas

Se escriben primero todas las pruebas. Convertir todos los elementos de la lista en pruebas antes de que pase ninguna es la interpretación errónea que destaca Beck. Si la primera obliga a reconsiderar una decisión, hay que rehacer todas las pruebas especulativas, y quien llega a la sexta sin haber visto pasar ninguna se encuentra, según escribe, con depresión o aburrimiento. Escriba una prueba, haga que pase y elija después la siguiente. Si una prueba nueva muestra que el trabajo realizado no puede atender un caso, decida entre continuar o empezar de nuevo; Beck aconseja empezar de nuevo y abordar los casos en otro orden.

La prueba no tiene ninguna aserción. Una prueba que llama al código y no comprueba nada aumenta la cifra de cobertura sin demostrar nada. Toda prueba necesita preparación, una invocación y una aserción, y Beck propone intentar de vez en cuando escribir una prueba al revés, empezando por sus aserciones.

Se elimina la aserción. Cuando una prueba no pasa, eliminar su aserción hace que finja pasar. Haga que pase de verdad o devuelva el elemento a la lista y reconsidere el orden.

Se copian los valores esperados de la salida. Ejecutar el código, copiar lo que devuelve y pegarlo en la prueba convierte una doble comprobación en una sola: la prueba ya solo confirma que el código hace lo que hizo antes. Calcule el valor esperado a partir del requisito, a mano si es necesario.

Se refactoriza mientras se intenta que la prueba pase. Cambiar la estructura y el comportamiento en el mismo paso es lo que Beck llama desempeñar dos papeles a la vez. Primero haga que funcione y después mejórelo: llegue al verde con el cambio más sencillo y ordene el código solo cuando las pruebas pasen.

Se ordena demasiado o se abstrae demasiado pronto. Ordenar el código resulta agradable y puede parecer más seguro que enfrentarse a la siguiente prueba. Pare cuando el código esté suficientemente bien para esta sesión y deje intactas dos partes parecidas hasta que el patrón esté claro. Beck lo resume así: «La duplicación es un indicio, no una orden».

Las pruebas conocen la implementación. Las pruebas que llaman a métodos privados o comprueban qué llamadas hizo el código, en lugar de qué produjo, se rompen cada vez que cambia su funcionamiento interno. El capítulo de Erik Kuefler sobre pruebas unitarias en *Software Engineering at Google* las llama pruebas frágiles: fallan ante un cambio no relacionado que no introduce ningún defecto real. En TDD el coste es doble, porque el paso de refactorización depende de que las pruebas sigan pasando mientras cambia el interior. Pruebe a través de la interfaz pública, compruebe el estado resultante en lugar de las interacciones y dé a cada prueba el nombre del comportamiento que comprueba.

Lecturas adicionales

- Canon TDD — exposición de Kent Beck de 2023 sobre el flujo de trabajo: los cinco pasos, la separación entre diseño de interfaz e implementación y los errores de cada paso.
- Simple Smalltalk Testing: With Patterns — copia archivada del artículo de Beck sobre un marco de pruebas sencillo para Smalltalk, con sus elementos de preparación, casos de prueba y conjuntos de pruebas.
- Pruebas unitarias — capítulo de Erik Kuefler de *Software Engineering at Google* sobre pruebas que resisten la refactorización, pruebas a través de API públicas y pruebas de comportamientos.
- Primeros pasos — introducción a pytest: instalación, primera prueba que falla con su informe y búsqueda de archivos de pruebas.
- unittest: marco de pruebas unitarias — marco de la biblioteca estándar de Python, que identifica el artículo de Beck como descripción original del patrón que comparte.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es el desarrollo guiado por pruebas?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/test-driven-development/