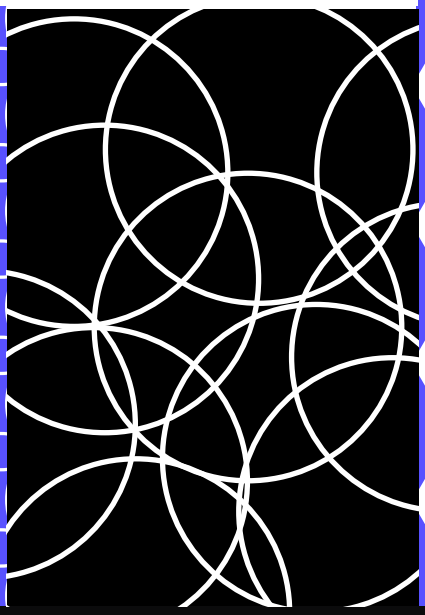
PATRONES COMUNES QUE DIFICULTAN EL TDD

Un ensayo de profesionales



Matheus Marabesi

codurance

PATRONES COMUNES QUE DIFICULTAN EL TDD

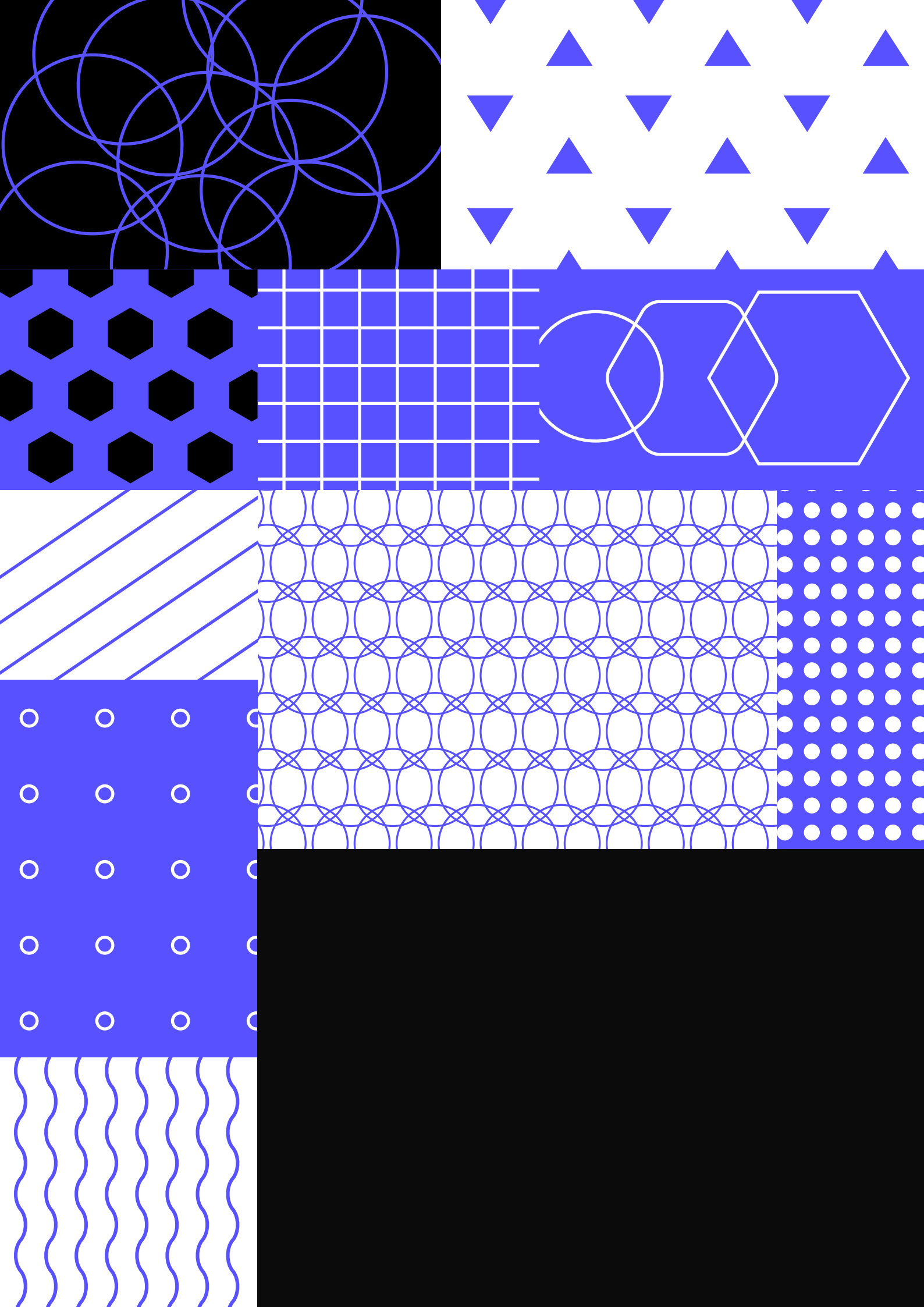
Un ensayo de profesionales

Matheus Marabesi
2023

Toda persona que desarrolla código sabe que debe escribir pruebas para su código. Pocos lo hacen.

Kent Beck y Erich Gamma

TABLE OF CONTENTS



Prólogo

Han pasado casi 20 años desde que Kent Beck presentó su “revisión” del *Test Driven Development* (o TDD, como se conoce más comúnmente) y publicó su posterior libro, *Test Driven Development: By example*.

Mi viaje comenzó mientras leía *Growing Object Oriented Software Guided By Tests*, de Nat Pryce y Steve Freeman. Recordando aquella época, no puedo evitar pensar que me hubiese sido muy útil tener este eBook sobre patrones de TDD de Matheus Marabesi. Muchos de los patrones que presenta los he experimentado en mi carrera profesional trabajando con diferentes equipos de desarrollo de software.

Desde la “revisión” de Kent Beck han surgido muchas publicaciones, conferencias, podcasts y otros materiales que destacan las virtudes del TDD. Sin embargo, son muchos menos los que se centran en aquellos patrones de testing que pueden ralentizar el desarrollo a lo largo del tiempo y dificultar una adopción más amplia.

Los patrones analizados y los ejemplos de código que los acompañan pueden utilizarse como guías para implementar TDD y mantener un conjunto de pruebas automatizadas. En ese sentido, este eBook se convierte en un recurso especialmente útil para maximizar los beneficios de TDD.

Matt Belcher
Principal en Codurance

Agradecimientos

Elaborar un eBook como éste ha sido todo un reto. Ha supuesto recopilar información de proyectos de código abierto y de profesionales que trabajan a diario con *test-driven development*. A lo largo de la creación de este contenido, muchas personas han contribuido aportando sus comentarios y compartiendo su tiempo, energía y conocimientos para ayudar a la creación de este contenido.

Me gustaría agradecer especialmente a Codurance el espacio y el esfuerzo dedicados a crear un entorno que fomenta el aprendizaje continuo y el intercambio con la comunidad.

Este libro es también el resultado de una serie de charlas que impartí en Codurance sobre anti-patrones de TDD y que me impulsaron a pensar en cómo podía crear una forma más estructurada de recopilar ese conocimiento. Un gran agradecimiento a Helena Abellán por todo su trabajo duro entre bastidores, recopilando toda la información necesaria y haciendo que la serie de vídeos fuera un éxito.

Además, un agradecimiento muy especial a todos los que participaron en la serie de vídeos y colaboraron: Cameron Raw, Giulio Perrone, Juan Pablo Blanco, Javier Martínez Alcántara, Sofía Carballo, Ignacio Saporitti y Pablo Díaz.

Matheus Marabesi
Software Craftsperson en Codurance

Contáctanos

Si detectas algún problema con el contenido o quieres hacernos llegar tus comentarios, sigue el enlace a este google forms.

Licencia

Prefacio

La idea de escribir código de prueba antes del código de producción (o, como algunos lo llaman, "código real")[1] provoca sentimientos encontrados entre los programadores experimentados. Esto se debe al pensamiento de que "no tiene sentido escribir código para probar código". En situaciones extremas, los profesionales pueden pensar que son lo suficientemente inteligentes o experimentados como para no perder el tiempo generando código de prueba.

Una vez superada esa primera reacción, la aceptación se produce pero gradualmente, ya que cambiar la forma en que alguien está acostumbrado a trabajar no es tarea fácil. Experimentar nuevas formas de actuar requiere esfuerzo y voluntad y, de un modo u otro, todos hemos pasado por ello. Escribir código y ver que funcionaba a la primera se consideraba un éxito.

Cuando se supera la primera fase de rechazo y se comprende que un enfoque basado en pruebas puede ayudar al flujo de desarrollo (al evitar la regresión), se llega a la etapa final: aceptar que escribir tests ayuda a largo plazo, ya que construye una red de seguridad en la creación de software.

La aceptación es sólo el primer paso en el camino hacia el dominio de TDD, un reto en sí mismo; pero a lo largo del camino, los profesionales se enfrentarán a muchos escenarios donde son necesarias las pruebas y se encontrarán con muchos obstáculos, compensaciones y posibles trampas.

Por ejemplo, es posible que se busque que un conjunto de tests se ejecute lo más rápido posible[2] para garantizar un bucle de retroalimentación corto. Por otro lado, se puede evitar que las pruebas sean muy largas y que pretendan probarlo todo en un solo caso de prueba.

---

1 El código de producción se utiliza para indicar el código que se ejecutará cuando los usuarios interactúen con la aplicación; los profesionales también podrían referirse a él como el código "real".

2 Ejecutando 1000 pruebas en 1s por @marvinhagemeist: https://marvinh.dev/blog/running-1000-test-in-1s.

En este sentido, en Codurance intentamos ofrecer una nueva forma de ver los distintos antipatrones a los que pueden enfrentarse los desarrolladores a la hora de escribir código testeable (o de intentarlo). Este libro es el resultado de esa iniciativa.

¿Quién debería leer este libro?

Este libro está dirigido a los profesionales que deseen explorar determinados patrones de *testing* que pueden dificultar el TDD y, en consecuencia, percibir que los tests no aportan valor al ciclo de desarrollo. A lo largo del contenido, trataré de exponer los inconvenientes que se esconden tras esa idea con ejemplos prácticos en los que veremos cómo evitarlos al escribir código de prueba o de producción. Como veremos, ambos están estrechamente relacionados.

En este libro nos dirigimos a aquellas personas que tienen cierta experiencia en escribir código dando prioridad a las pruebas, pero que se sienten identificados con estas ideas de un modo u otro:

- No perciben el valor de las pruebas
- Dedican más tiempo a depurar las pruebas que a utilizarlas como guía
- Esperan demasiado para recibir información de las pruebas
- Perciben disparidad entre la retroalimentación de las pruebas y el código de producción

Sin embargo, incluso si tienes poca experiencia previa escribiendo código de prueba, descubrirás algo valioso que sacar de este libro. ¡Estás de suerte! porque vas a encontrar patrones a evitar cuando escribas tests en el futuro. Aclarar que este libro no está dirigido a profesionales que ya conocen los antipatrones de TDD, ni cómo abordar escenarios difíciles para progresar mientras se desarrolla código guiado por tests. Sin embargo, si perteneces a esta categoría, puede que encuentres algo útil leyendo los capítulos en diagonal.

Aunque se ha intentado que el contenido sea lo más sencillo posible para quienes se inician en TDD, se muestran diferentes patrones, lenguajes de programación y *frameworks*. Cada capítulo ofrece recursos adicionales para profundizar en un tema concreto si es necesario.

Quizá hay partes del contenido que te resulten exigentes; en ese caso, para sacar el máximo partido de este libro, te recomendamos que comprendas los conceptos y contenidos que se indican a continuación antes de seguir leyendo. En resumen, a continuación te exponemos algunos conceptos básicos que necesitas conocer y comprender para sacar mayor provecho a la lectura del libro.

- Programación orientada a objetos
- Desarrollo web (HTML, JavaScript y los respectivos frameworks como VueJs y ReactJs)

- Frameworks de testing como Junit, PHPUnit, Jest.

Este libro no pretende ser una recomendación de “mejores prácticas”[3], sino un enfoque hacia ellas. Lo que queremos evitar es esa falsa idea de que “una solución vale para todo”. Lo que sí pretende ser es una descripción ajustada de nuestras sugerencias, basadas en la experiencia de llevar unos cuantos años practicando TDD a las que hemos sumado los resultados de una encuesta que hemos realizado entre diferentes profesionales del sector para conocer la opinión de otros desarrolladores.

También quiere ser una manera de contribuir y compartir conocimiento con toda la comunidad. Es posible que en tu caso concreto hayas vivido situaciones diferentes que no estén incluidas en este libro, pero como se indica en el párrafo anterior, se trata de una recopilación de nuestras experiencias y las de quienes participaron en la encuesta.

Por último, si lo que esperas encontrar es una guía exhaustiva de patrones paso a paso, al estilo receta de cocina, sentimos decirte que este no es el contenido que andabas buscando.

Introducción

Uno de los mayores retos para los programadores es seguir con el enfoque de *test-first*. A menudo, los proyectos no están preparados para las pruebas y los equipos carecen de una cultura de *testing* sólida, por ello las pruebas automatizadas son escasas o inexistentes (ya se trate de TDD[4], TLD[5] o cualquier otro enfoque). Accelerate (Radziwill 2020) compartió que los equipos de alto rendimiento utilizan la automatización, y que TDD es una práctica importante para que sea posible.

En el panorama general del desarrollo de software, el enfoque de *test-first* está ‘en pañales’ en comparación con cómo se construía el software antes. La premisa era que el software siempre iba a funcionar, por lo que no era necesario realizar pruebas previas. Se escribía el código de producción, se compilaba y se probaba. Si funcionaba, estupendo, quizá se hacían algunos casos más, y entonces se podía seguir adelante. Normalmente ese era el proceso a seguir.

---

3 Puesto que las “mejores prácticas” dependen del contexto, puede ser más un obstáculo que una ayuda. En su lugar, utilizamos el término “prácticas sensatas por defecto”, descrito en el podcast sobre tecnología de Thoughtworks: *Starting with sensible default practices* available at https://www.thoughtworks.com/en-es/insights/podcasts/technology-podcasts/sensible-defaults.

4 Test Driven Development

5 Test Last Development

Algunos profesionales tuvieron que crear la cultura ellos mismos, forjando la mentalidad desde la base en sus equipos.[6] La idea suele ser buena, pero puede resultar difícil de mantener a largo plazo. Por ejemplo, si estás intentando impulsar un estilo de trabajo alternativo y tus compañeros no reconocen su valor, entonces, sin el apoyo necesario, lo más fácil sería abandonar la causa y seguir con el hábito de escribir código y completar pruebas manuales. Al final, es más fácil volver al estilo de desarrollo al que todos estamos acostumbrados para construir aplicaciones.[7]

Por otro lado, diferentes equipos han adoptado el *Test Driven Development* o Desarrollo Dirigido por Pruebas (TDD) como una forma de entregar nuevas funcionalidades, impulsar bucles de retroalimentación más cortos y evitar regresiones en las funciones existentes.

Kent Beck (2003) popularizó esta metodología que se convirtió en un estándar en la industria. Desde entonces, otros la han mejorado y evolucionado sobre ella. Iniciarse en TDD no es fácil y requiere de un mantenimiento constante para garantizar, por ejemplo, que un conjunto de pruebas sea capaz de ejecutarse con rapidez. Las bases de código que abordan problemas de negocio requieren una gran cantidad de código y, con él, un conjunto de casos de prueba considerable.

Para abordar las necesidades que dicta el negocio, se requieren diferentes tipos de pruebas. La pirámide de pruebas[8] descrita en el libro *Successing with Agile* de Mike Cohn (2009), y posteriormente referenciada por Vocke (2018), sugiere lo siguiente:

1. Para empezar disponer de una base sólida de *tests* unitarios que idealmente se ejecuten lo más rápido posible y proporcionen una retroalimentación rápida.
2. En mitad del proceso tenemos los test de integración que pueden ser más lentos que los unitarios pero proporcionan *feedback* si las piezas más pequeñas funcionan como deberían.
3. Por último, tenemos los tests *end-to-end* (representados como UI Tests), también conocidos como pruebas que actúan como si fueran un usuario (ya sea un humano u otro sistema/programa).

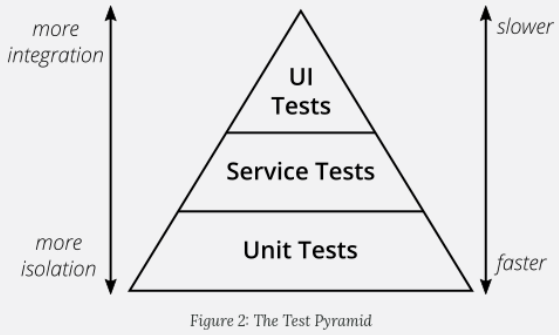
Según los datos recogidos que hemos recogido para elaborar en este ebbok, parece ser que el planteamiento de tener el conjunto de pruebas con esta forma piramidal no es lo más común en la mayoría de los proyectos que se llevan a cabo en la realidad.

---

6 Julio César Pérez compartió su experiencia frente a este escenario, donde relató que los miembros de su equipo se resistían a adoptar un enfoque “test-first”. Puedes leer el blog en: https://www.codurance.com/es/publications/una-historia-de-testing.

7 A menudo se confunde TDD con código libre de errores, lo que no es necesariamente correcto (Dijkstra y otros, 1970).

8 Vale la pena mencionar que esta misma pirámide fue citada por Ham Vocke en el blog de Martin Fowler https://martinfowler.com/articles/practical-test-pyramid.html.

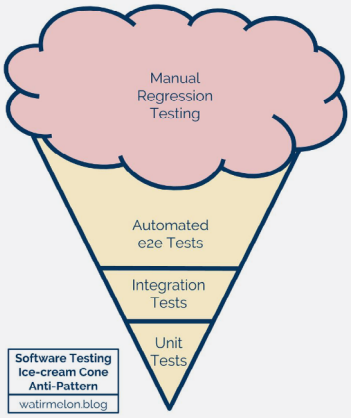


El Test Pyramid de Mike Cohn (2009)

Los patrones aquí analizados sugieren que hay un concepto algo erróneo sobre cuál es la mejor manera de dividir los tipos de *tests* y sus responsabilidades. Por ejemplo, a menudo se hace referencia a la prueba unitaria como una relación de uno a uno entre la prueba y el código de producción[9].

Sin embargo, lo que nos encontramos más habitualmente en el sector es lo contrario. Por lo general (a pesar de que la pirámide de pruebas antes mencionada es una vieja conocida), las *suites* de pruebas automatizadas suelen estar compuestas por un mayor número de *tests* más lentos que rápidos. En efecto, más tests de integración y *end-to-end* que pruebas unitarias. Esto nos lleva al patrón *Ice Cream* en lugar de a la *Test Pyramid*.

El problema de que se esté aplicando el patrón *Ice Cream* en lugar de la pirámide es el malestar que genera en los profesionales a la hora de mantener esas pruebas y, al mismo tiempo, la tasa de entrega.



El cono de helado, también en https://alisterbscott.com/kb/testing-pyramids/#site-header

9 En los casos de prueba que se escriben teniendo en cuenta la Programación Orientada a Objetos, una clase de prueba significa una clase de producción.

Además, Wang, Pyhäjärvi y Mäntylä (2020) describen que en la industria el proceso de automatización de pruebas aún es inmaduro, lo que se traduce en una retroalimentación lenta. Por un lado, sabemos que testear el software es importante y necesario, pero, por otro, el sector no está adoptando procesos de *testing* eficaces y/o eficientes.

Otro tema que se observa es el concepto erróneo respecto a la cobertura del código10, que es una métrica que los directivos suelen utilizar para forzar a los equipos de desarrollo a escribir pruebas automatizadas.

Esta métrica por sí sola no es una buena medida del éxito o de que se estén aplicando buenas prácticas de *testing*. Por ejemplo, los profesionales pueden decidir escribir pruebas automatizadas de tal manera que las pruebas ejerciten el mayor número de líneas de código con el menor número de pruebas. En lugar de seguir un enfoque de TDD con un gran número de *tests* más pequeños. Cuando se les pregunta a los equipos de desarrollo sobre esta estrategia es habitual que no estén de acuerdo con ella.

En este sentido, más adelante analizaremos un antipatrón que puede ocurrir a través de esta forma de perseguir la cobertura de pruebas.

Esto no significa que tener en cuenta esta métrica sea malo por sí mismo, sino que tiene su propio lugar en el proceso de desarrollo, como sugiere Mauricio Aniche (2022).

Centrándonos en el tema que aquí nos interesa, aunque esto parezca algo muy nuevo, ya hace más de diez años que se hicieron esfuerzos por reunir algunos de esos inconvenientes que sufren los equipos de desarrollo cuando intentan escribir código guiado por pruebas. James Carr (2022) ideó una lista de antipatrones a tener en cuenta para evitar el patrón del *Ice Cream Cone* en el que pueden caer extensas bases de código.

Posteriormente, se creó un hilo de votación en StackOverflow[11] para abrir el debate y que otros profesionales puedan contribuir a la lista.

Más recientemente, Dave Farley también repasó algunos de ellos en su canal de YouTube en un vídeo que denominó "When Test Driven Development Goes Wrong" (Farley 2021) en un intento de describir los problemas que podemos encontrar al practicar TDD de una manera inadecuada. La mayoría de los ejemplos utilizados proceden de proyectos de código abierto y son ejemplos ficticios basados en situaciones reales. Esta estrategia fue deliberada y se utilizó para dar al espectador una idea general de los antipatrones y representar que tales escenarios ocurren a diario.

---

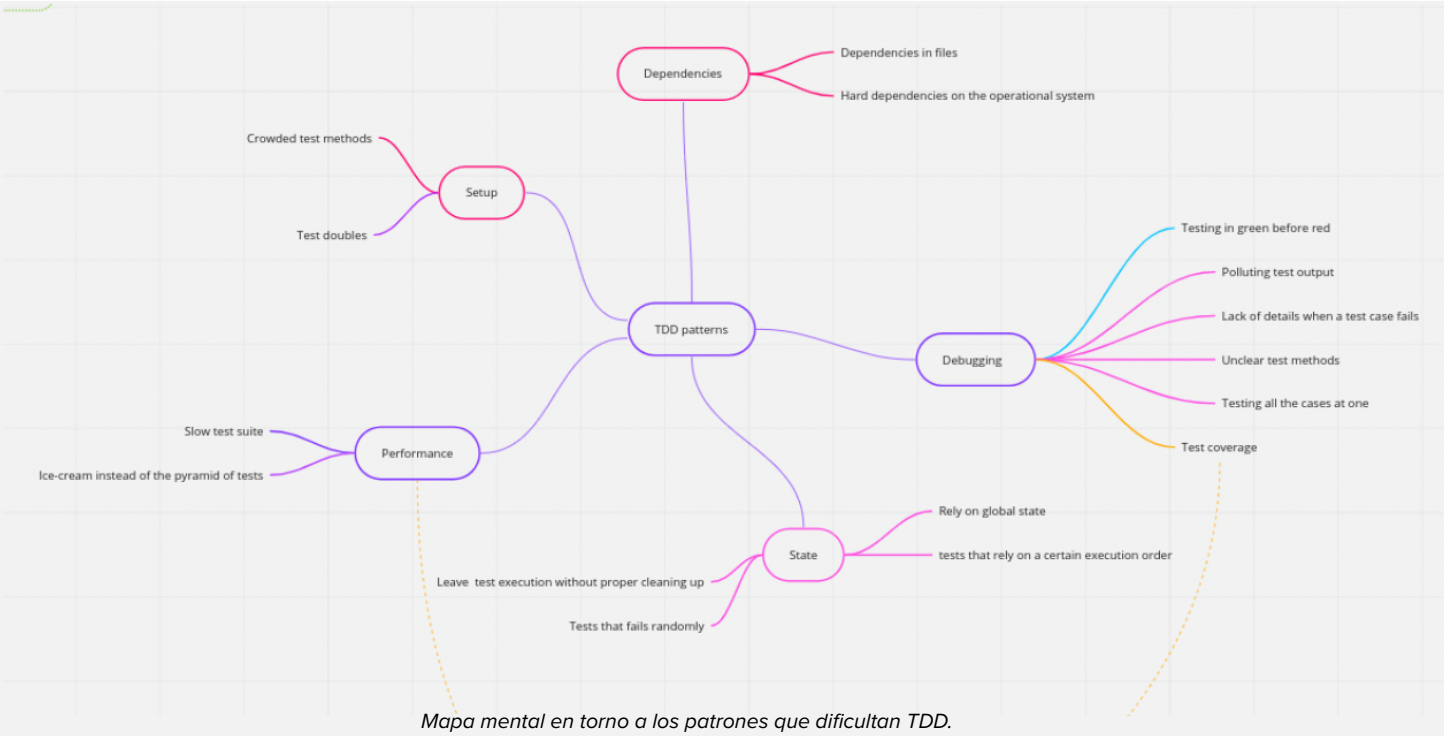
10 https://marabesi.com/thoughts/2021/05/29/on-100-percent-code-coverage

11 https://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue

A pesar del gran contenido presentado por Farley, los ejemplos no eran una lista exhaustiva, ni se basaban en la comprensión que tienen los profesionales sobre los antipatrones de TDD. En cambio, era su interpretación de los problemas encontrados cuando no se practica TDD de la "manera apropiada" (de ahí el título *When Test Driven Development Goes Wrong*).

Yegor Bugayenko (2021) también intentó analizar el campo de los antipatrones de TDD, y añadió algunos más a la lista original de James Carr. Esto preparó el camino para este libro, que ofrece ejemplos completos para cada antipatrón de la lista. Además, presenta dos antipatrones adicionales basados en la experiencia de observar a programadores que utilizan TDD a diario.

Durante este recorrido, se creó este mapa conceptual para visualizar los temas que vamos a tratar.[12] Este gráfico puede ser útil para ver hasta qué punto los antipatrones están relacionados con otras áreas mientras se practica TDD.



Mapa mental en torno a los patrones que dificultan TDD.

Estructura del libro

El libro está dividido en cuatro niveles (I–IV) que agrupan los antipatrones. Cada nivel ha sido pensado para representar el progreso de una persona que da sus primeros pasos en TDD; de ese modo el nivel I aborda problemas potenciales

12 Puedes verlo e interactuar con él accediendo al siguiente enlace https://bit.ly/3yumaBI.

para quienes se inician en el camino de TDD, mientras que el nivel IV se extiende a patrones más avanzados a medida que evoluciona la práctica de escribir tests.

Por supuesto, esta no es una regla fija, ya que puede darse el caso de que haya algunos profesionales podrían haberse enfrentado a problemas en el nivel I cuando ya tenían cierta experiencia escribiendo tests, lo cual también está bien.

La finalidad de los distintos niveles es puramente expositiva. Hemos formateado el contenido de manera que pueda seguirse de forma estructurada y permita a los lectores relacionarlo fácilmente con sus prácticas cotidianas.

Una nota antes de empezar

Antes de sumergirte en el libro, recuerda que el código y los ejemplos utilizados tienen únicamente fines educativos y no pretenden "hacer sentir culpable" o representar cosas que no deberían hacerse. Tener una base de código con más de uno de los patrones mencionados es habitual, y si aún no has visto ninguno de ellos ya llegará el momento.

El objetivo es arrojar luz, mediante la concienciación, sobre cómo gestionar estos posibles patrones que dificultan la verificación del código.

En este capítulo, revisaremos datos recopilados de profesionales que trabajan a diario en proyectos de software de todo el mundo.

La encuesta

Antes de seguir adelante, es importante hacer algunas aclaraciones sobre la encuesta que se ha realizado y su significado para este libro.

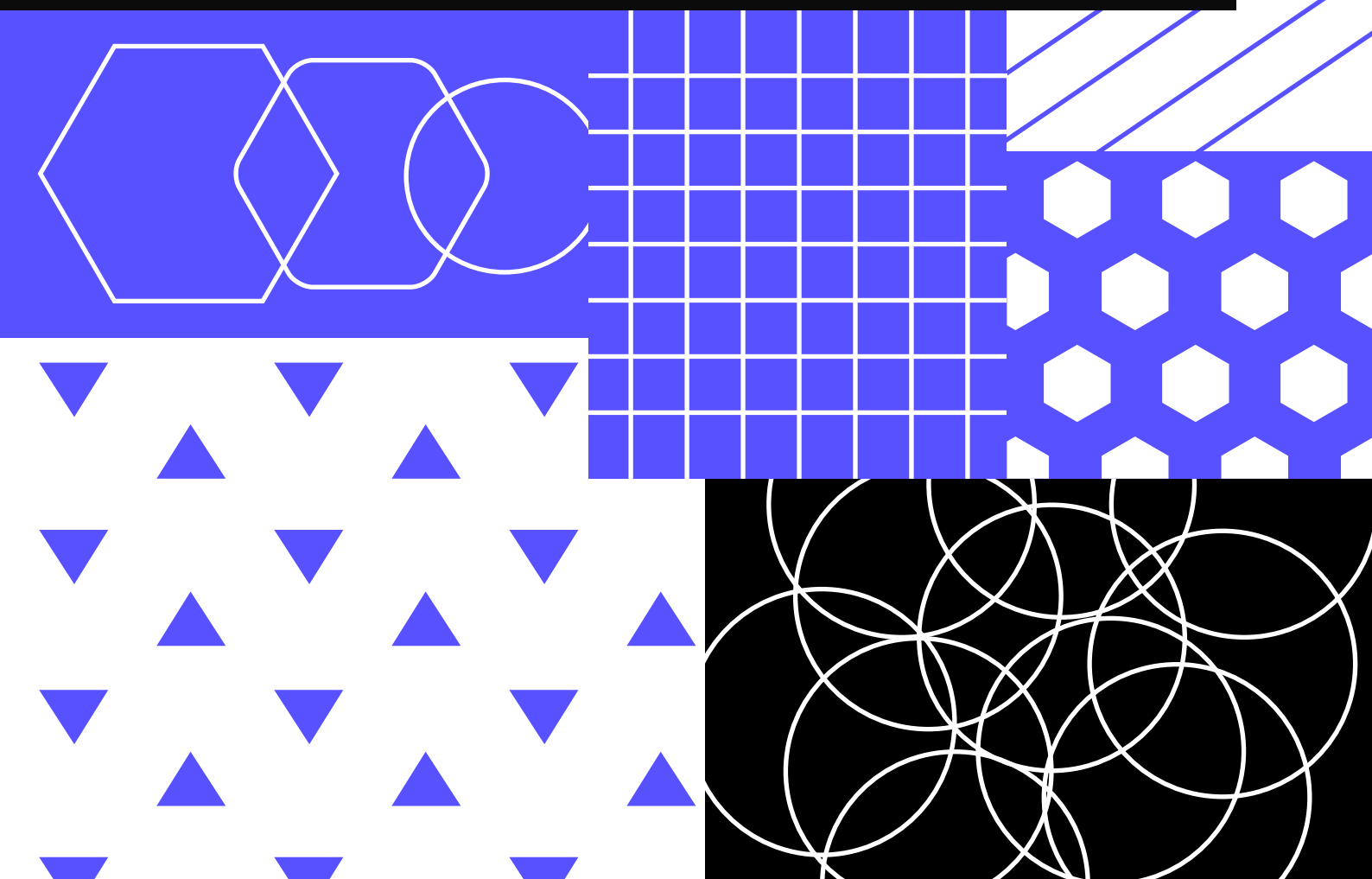
En primer lugar, queremos clarificar que es un sondeo que carece de relevancia científica o estadística para el tema estudiado. En otras palabras, los datos recogidos no pretenden servir de base para ningún análisis más allá de la recopilación de respuestas de los profesionales.

Como veremos en la siguiente sección, la metodología aplicada necesita de un proceso formal aunque sea reproducible.

Además, aunque los datos recopilados en bruto están disponibles en GitHub[13] como gist, hay que tener en cuenta que no se incluyen datos sensibles como las direcciones de correo electrónico.

13 https://gist.github.com/marabesi/5f0eafd3ea948a5c1dcd25720299ac17

Encuesta sobre los antipatrones de TDD



Encuesta sobre los antipatrones de TDD

Metodología

Antes de comenzar las charlas del circuito sobre antipatrones de TDD, presentado en Codurance, preguntamos a varios desarrolladores si querían participar en una encuesta que se centraría en recopilar datos para las siguientes sesiones.[14]

22 personas respondieron afirmativamente a la solicitud y nos ayudaron con información sobre sus actividades e intereses en relación con los antipatrones de TDD.

A partir de ahí, se creó una encuesta que contenía las preguntas que se adjuntan más adelante. El formulario se creó con google forms[15] y estuvo a disposición del público desde el 15 de septiembre de 2021 hasta el 5 de octubre de 2021[16]. Las respuestas registradas se computaron en septiembre de 2021, la primera el mismo día en que la encuesta estuvo disponible y la última el 27 de septiembre de 2021.

Se utilizó Twitter y LinkedIn para difundir la encuesta con el fin de compartirla con el mayor número de personas posible.[17]

La encuesta se estructuró en cuatro secciones destinadas a recoger datos en los siguientes ámbitos:

- Experiencia profesional
- Prácticas de TDD en el día a día
- Prácticas de TDD en empresas en las que he trabajado
- Antipatrones
- Datos personales

En la siguiente sección examinaremos los resultados y las conclusiones de esta encuesta.

---

14 Tweet enviado solicitando la participación de los profesionales en la encuesta https://twitter.com/MatheusMarabesi/status/1437525252121182214

15 https://www.google.com/forms

16 Estas fechas no son absolutas, pueden variar. La fecha de inicio es la más fiable (ya que se compartió en Twitter), en cambio, la fecha de cierre es la que carece de una comprobación formal ya que google forms no permite ver cuando un formulario se ha cerrado. Para más información sobre la fecha de inicio y cierre, consulte esta línea de tiempo de twitter https://bit.ly/3AhlBwt.

17 Como la difusión se limitó a Twitter y Linkedin, no llegamos a más encuestados potenciales. Esto supone una limitación de datos que debe tenerse en cuenta a la hora de analizarlos.

Resultados

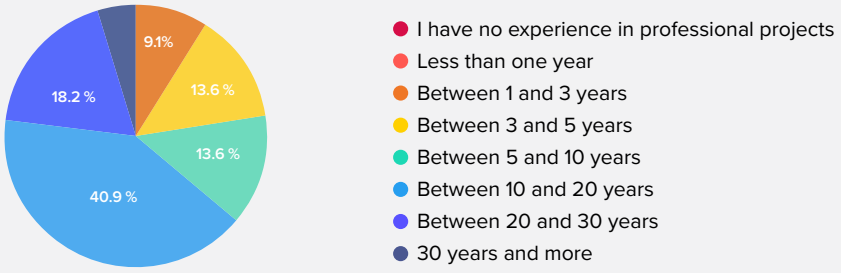
A continuación examinamos los resultados de la encuesta. Seguiremos la misma estructura de secciones mencionada anteriormente, empezando por los resultados de *Experiencia profesional*, seguidos de los resultados de *Prácticas de TDD en el día a día, Prácticas de TDD en empresas en las que he trabajado y Antipatrones*.

Experiencia Profesional

El objetivo de la encuesta era recopilar datos sobre los antipatrones de TDD en la industria. Obtuvimos 22 respuestas, y los datos muestran que los profesionales encuestados trabajaban en proyectos en América Latina: Argentina, Brasil y México, y Europa: España, Francia, Hungría, Irlanda, Portugal y Rumanía.

El 60% de los encuestados trabaja en proyectos profesionales del sector, (sin tener en cuenta los años de experiencia) un 60% y todos ellos afirman contar con experiencia probada (el 22,7% tiene entre 1 y 5 años de experiencia)

El 40,9% cuenta con entre 10 y 20 años de experiencia trabajando en proyectos profesionales.

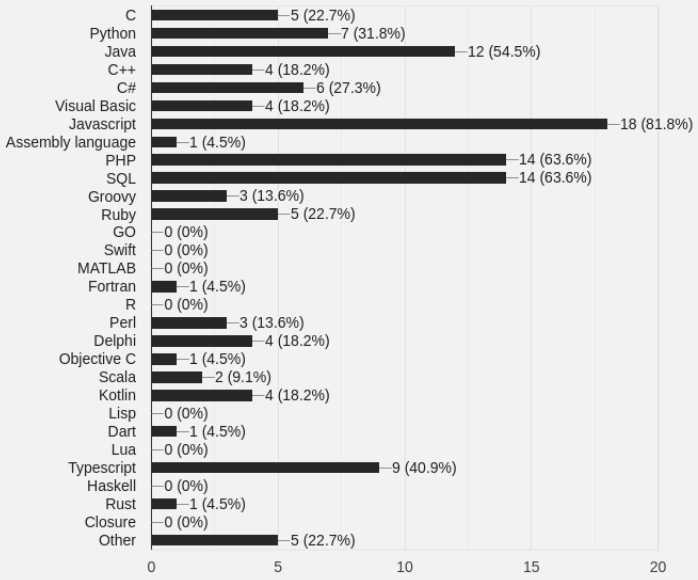


Respuestas a la pregunta: "Soy desarrollador de software y trabajo en el sector profesionalmente".

También hemos descubierto que los lenguajes de programación más populares son:

- JavaScript
- PHP
- Java/Kotlin
- Typescript
- Python

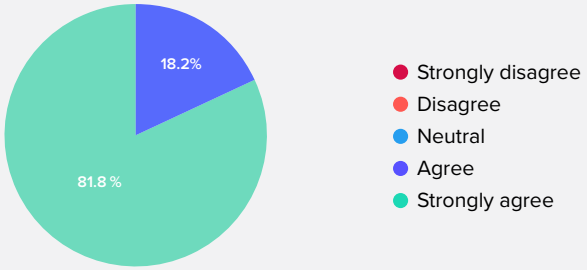
A pesar de que vemos una variedad de lenguajes utilizados:



Respuestas a la pregunta "Escribo/escribí código profesionalmente en los siguientes lenguajes". (Los lenguajes de programación enumerados son de https://www.tiobe.com/tiobe-index).

La mayoría de los ejemplos utilizados en las secciones siguientes también están en JavaScript. Creemos que así será más fácil llegar a un público más amplio y devolver el favor a los que respondieron a la encuesta. Por otro lado, también se muestra que los lenguajes menos populares son: Ruby, Rust y Groovy.

Observamos que los encuestados están familiarizados con diferentes herramientas de pruebas como Junit, Jest, PHPUnit o cualquier otro marco que proporcione una base común para escribir pruebas. Cabe destacar que todos respondieron tener algún conocimiento sobre estas herramientas.



Respuestas a la pregunta: "Estoy familiarizado con herramientas de pruebas como Junit, Jest, PHPUnit, o cualquier otro framework que proporcione una base común para escribir pruebas".

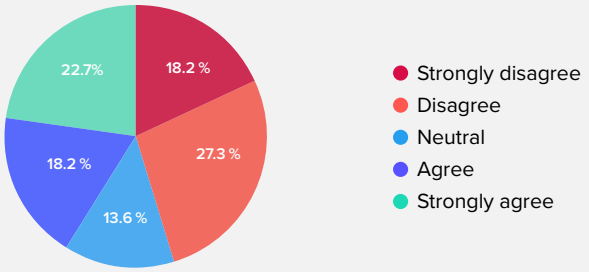
Si volvemos a la primera pregunta de esta encuesta, vemos que son muy pocas

personas las que afirman no trabajar profesionalmente en proyectos, pero que, aun así, están familiarizadas con las herramientas de tests disponibles en el *open source*.

Prácticas de TDD en el día a día

A continuación, examinaremos las prácticas de las personas que utilizan el enfoque test-first al desarrollar aplicaciones.

Dado que la mayoría de los encuestados trabajan en proyectos profesionales, la siguiente pregunta pretende abordar cómo aprendieron a aplicar TDD, y en concreto si lo habían aprendido en el trabajo. La mayoría de las personas (45,4 %) indicaron que no era su caso, y hay un 27,3 % que se muestran en desacuerdo y un 18,2% 'totalmente en desacuerdo' lo cual nos indica que el lugar de trabajo no ha sido el espacio en el que la gran mayoría ha aprendido TTD.



Respuestas a la pregunta: "Aprendí TDD en el trabajo".

La siguiente pregunta se enfoca en el modo en el que los profesionales, que no aprendieron TDD en su trabajo, conocieron esta práctica. Un 86,4% indicó haber aprendido TDD por su cuenta y a través de libros, videos o tutoriales.

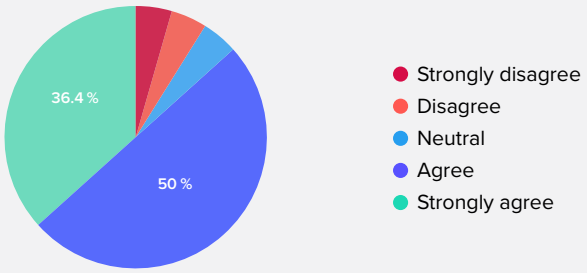
Esto quiere decir que aprendieron TDD "informalmente"[18], lo que significa que, a partir de sus respuestas, más del 50% de ellos aprendieron TDD solo a través de vídeos, libros o tutoriales.

En este sentido sólo el 50% de los sondeados afirman que creen que las empresas comprenden los pros y los contras del TDD y lo practican. Es importante resaltar que estos datos proceden de las respuestas obtenidas por los profesionales y, por ende, se trata de la percepción que tienen sobre el tema.

Esta pregunta no especifica la combinación de todas las opciones dadas. Es posible que los profesionales mezclaran distintas formas de

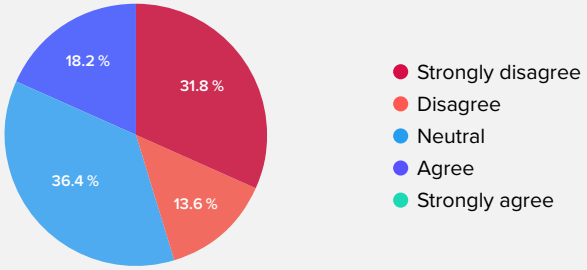
18 Con 'informalmente' nos referimos a que no poseen una certificación oficial y ninguna institución corroborará sus conocimientos sobre TDD.

aprender TDD. Por ejemplo, viendo videos online y leyendo un libro para rellenar algunas posibles lagunas.



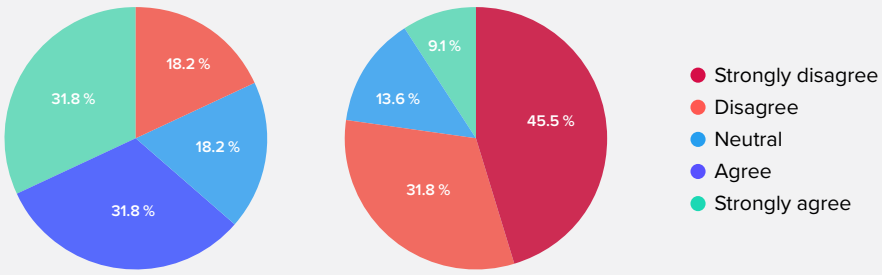
Respuestas a la pregunta: "Aprendí TDD por mi cuenta, a través de libros, vídeos o tutoriales".

También preguntamos a los sondeados si sus compañeros de trabajo conocían TDD. La mayoría de ellos (45,4 %) respondió que no, mientras que el 36,4 % se mantuvo neutral.



Respuestas a la pregunta: "La gente con la que trabajó/trabajaba conocía TDD".

En relación con sus contextos profesionales, también preguntamos si sus compañeros practican TDD diariamente y si trabajan escribiendo tests-first.



A la izquierda, verán las respuestas a la pregunta: "Las personas con las que trabajo practican TDD a diario". A la derecha, las respuestas a la pregunta: "Práctico el TDD a diario".

Adicionalmente, preguntamos si percibían que su trabajo se ralentizaba al adoptar TDD. Esto suele ocurrir cuando se está empezando a aprender a probar código. De hecho, sucede cada vez que se aprende una hard skill nueva. Si lo pensamos, la fase inicial de cualquier tecnología nueva es a la que más nos cuesta acostumbrarnos.

El 77,3 % de los encuestados no está de acuerdo con esta afirmación. Es decir, la mayoría de los profesionales creen que practicar TDD es valioso y no sienten que les ralentice[19].

En este sentido, Uncle Bob, conocido autor del libro Clean Architecture (Martin 2017), cuando hablaba sobre el uso de TDD afirmó que "la única manera de ir rápido es ir bien". Tal afirmación se refiere a la percepción de que TDD ralentiza el flujo del desarrollador. Esto se debe a que el beneficio de testear código no es visible inmediatamente después de escribir la prueba, sino en la refactorización. Es en esta fase que las pruebas detectan posibles fallos que se habrían notado sólo en la producción por el usuario final.[20]

Prácticas de TDD en las empresas en las he trabajado

A continuación, examinaremos más preguntas relacionadas con el entorno cotidiano en el que trabajan los profesionales.

Empezamos con la pregunta, "No se me permite enviar código para su revisión sin un caso de prueba". Para ello, asumimos que los practicantes trabajan de modo gitflow[21] en lugar de Trunk Base Development[22] [23]. El pull request es un método que los profesionales utilizan habitualmente para pedir revisiones de código.

La mayoría de los profesionales (63,6 %) están de acuerdo con esta declaración. Sin embargo, es necesario profundizar más para entender a qué se debe: ¿se ven obligados por alguna jerarquía? ¿Por qué existe tal restricción?

---

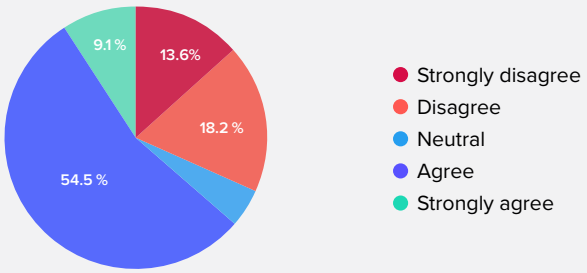
19 Es importante mencionar que esta encuesta se compartió a través de Twitter y LinkedIn para que llegará a aquellas comunidades que ya están familiarizadas con TDD. Esto se debe tener en cuenta ya que los resultados de la encuesta no representan el estado de la industria, sino más bien un pequeño número de profesionales que trabajan en ella.

20 La "lentitud" o "rapidez" puede estar relacionada con la propia codebase, pero normalmente se utiliza como argumento para evitar el uso de TDD.

21 https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

22 Dave Farley también ha compartido su opinión sobre por qué git-flow podría ser una mala idea https://www.youtube.com/watch?v=_w6TwnLCFwA

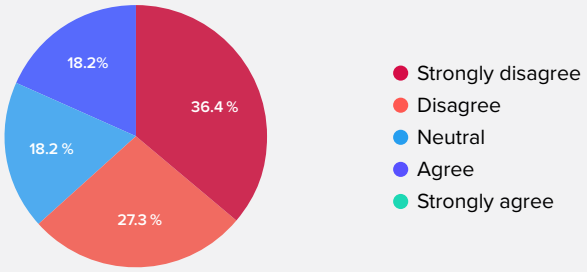
23 Descubre qué es el Trunk Base Development en este enlace https://trunkbaseddevelopment.com



Resultados de la pregunta: "No se me permite enviar código para su revisión sin un caso de prueba".

También preguntamos si las empresas para las que trabajaban exigían TDD para formar parte de los equipos. Esta vez, vemos que un 18,2% seleccionó 'neutral', lo que puede significar que algunas personas no trabajan para ninguna empresa.

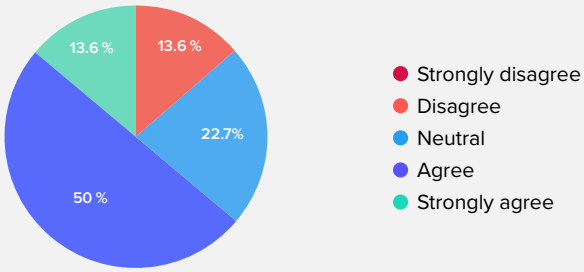
Si combinamos "totalmente en desacuerdo" y "en desacuerdo", vemos que la mayoría de las empresas (63,7 %) no exigen TDD de antemano.



Respuestas a la pregunta: "Las empresas en las que trabajo/trabajé exigían que el TDD formará parte de la descripción del puesto".

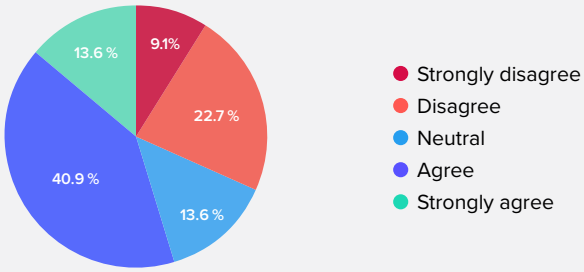
La siguiente pregunta trata de describir el entorno laboral respecto a la práctica del TDD. De la misma manera en la que la pregunta anterior demostraba que la mayoría de las empresas no exigen TDD como parte del trabajo, aquí vemos una potencial continuación de esa tendencia.

La mayoría de los profesionales (63,6 %) afirmó que la empresa para la que trabajan no practica TDD.



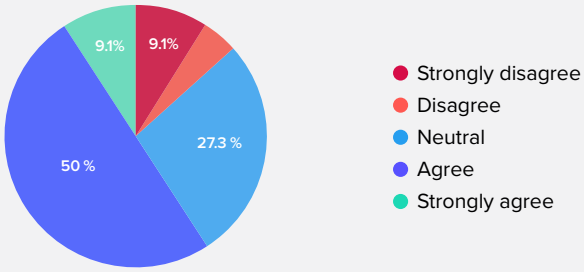
Respuestas a la pregunta: “Las empresas en las que trabajo/trabajé no practican el TDD”.

Asimismo, el 54,5 % de los encuestados compartieron que las empresas para las que trabajan creen que las tareas tardan más tiempo en ser completadas si el equipo utiliza TDD.



Respuestas a la pregunta: “Las empresas en las que trabajo/trabajé argumentaban que completar una tarea con TDD requiere más tiempo, y que los equipos no disponen del tiempo necesario para aplicarlo.”

Adicionalmente, preguntamos si la empresa en la que trabajaban conocía los pros y los contras de utilizar TDD, y el 59,9 % respondió que sí.



Respuestas a la pregunta: “Las empresas en las que trabajo/trabajé valoran la práctica del TDD y reconocen sus pros y sus contras.”

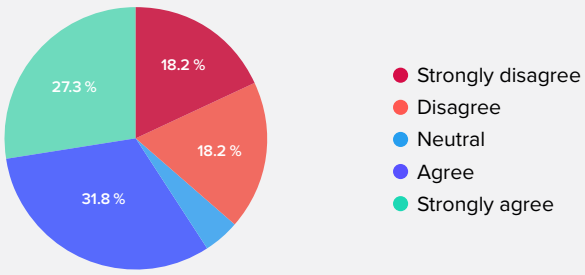
Esto puede relacionarse a la pregunta anterior e interpretarse como una decisión deliberada de utilizar o no TDD en el proceso de desarrollo de software o en el proceso de contratación.

A continuación, repasaremos la sección en la que le preguntamos a los profesionales sobre los antipatrones de TDD. Aunque la encuesta está compuesta por cuatro secciones, esta es la última sección utilizada para recopilar datos sobre la temática de este libro. La última sección, llamada Finishing up, recopila datos personales.

Anti-Patrones

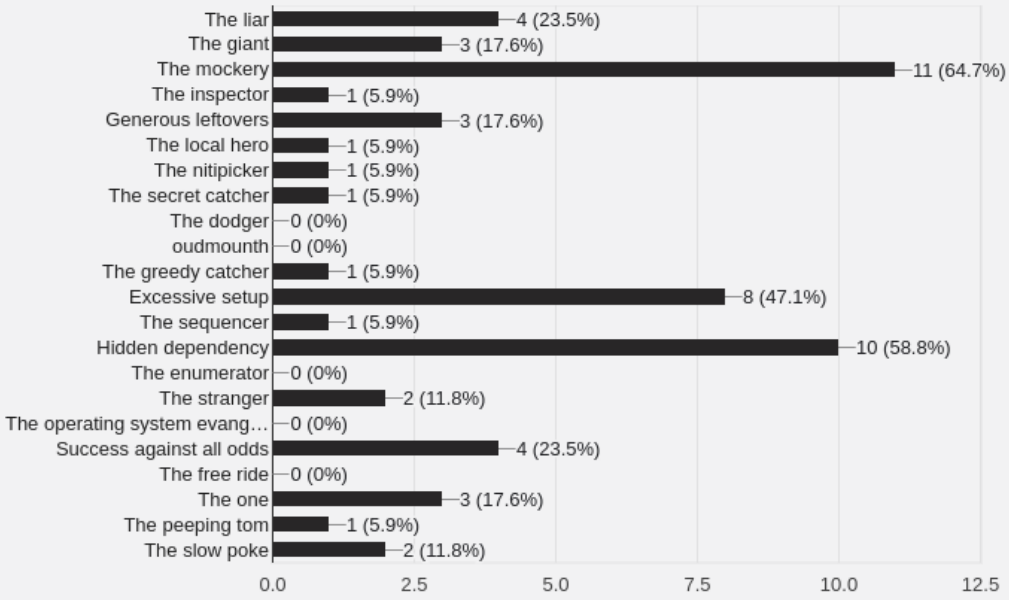
Esta sección está dedicada a lo que conocemos como “antipatrones de TDD”. Debido a que no son un tema habitual entre los profesionales, el objetivo de esta sección es describir qué saben realmente los profesionales sobre antipatrones y en qué medida.

Para empezar, la primera pregunta pretendía ver si los profesionales podían recordar al menos un antipatrón definido por James Carr (2022). A pesar de que los nombres de los antipatrones catalogados no son tan conocidos, esta pregunta demuestra que los profesionales pueden recordar algunos de ellos.

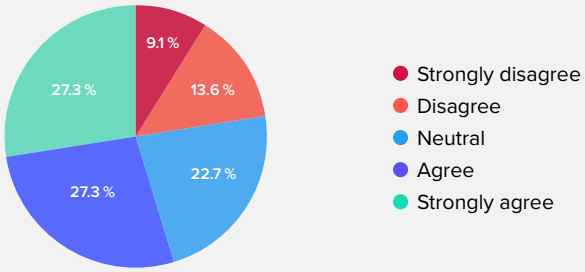


Respuestas a la pregunta: “Puedo recordar al menos un antipatrón de TDD”

De las respuestas obtenidas, el 54,6 % dijo que podía recordar al menos un antipatrón. La pregunta sucesiva se creó para demostrar qué antipatrones podían recordar los profesionales encuestados. El más popular fue el Mockery.



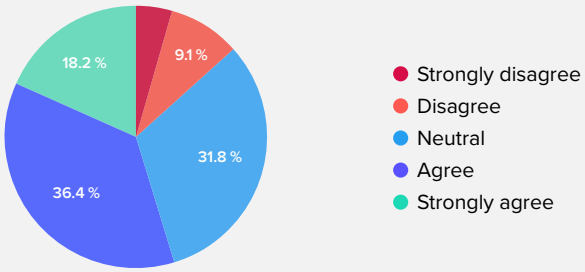
Respuestas a la pregunta: "De la siguiente lista, marca los antipatrones que recuerdas".



Respuestas a la pregunta: "Podría detectar al menos un antipatrón de TDD en el codebase en el que trabajo profesionalmente."

Uno de los aspectos conflictivos de los antipatrones es que, debido a la falta de concienciación, muchos de ellos no se detectan a diario.

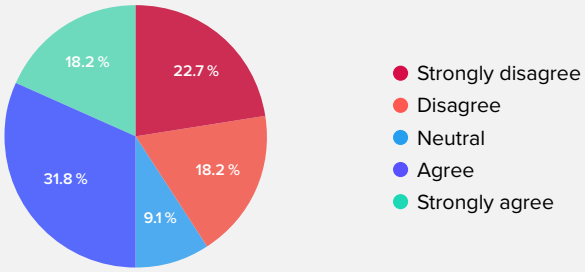
Respecto a la pregunta "Creo que los antipatrones de TDD me ralentizan", el 31,8 % de las personas se mantuvieron 'neutrales'. Es posible que esto esté relacionado con la pregunta anterior "puedo recordar al menos un antipatrón de TDD" que tuvo un 22,7 % de respuestas. Pues, si no pueden recordar los antipatrones es difícil que identifiquen si les ralentiza o no.



Respuestas a la pregunta: "Siento que los antipatrones de TDD me ralentizan".

La siguiente pregunta se refiere a algunos antipatrones que surgen cuando el test se escribe después de la producción (si es que se escribe en lo absoluto), lo que genera una codebase sin pruebas.

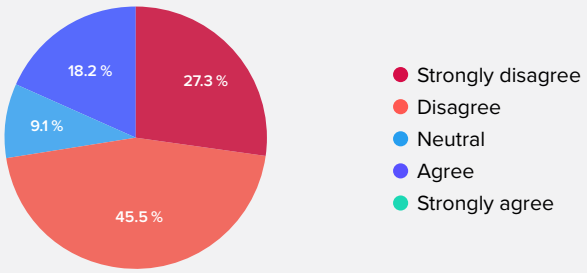
En este sentido, cuando los profesionales intentan añadir algunas pruebas, se retrasa el proceso de hacer que algo funcione con el test primero. En total, el 50% de los profesionales está de acuerdo con esto.



Respuestas a la pregunta: "Cuando intenté practicar TDD en un codebase sin pruebas, sentí que me atrasó escribir la prueba primero".

La cobertura de código suele utilizarse para explorar el codebase y evaluar dónde hacen falta más casos de prueba. Como tal, esta suele ser la forma utilizada.

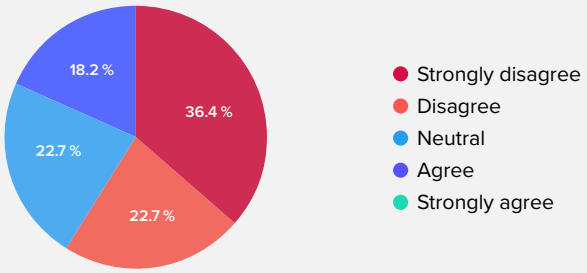
De los profesionales que respondieron, el 18,2% se preocupa por la cobertura. Este dato contrasta con el 72,8% que no está de acuerdo. Este suele ser un tema que divide a los profesionales ya que puede usarse de manera ineficaz.



Respuestas a la pregunta: "Escribo pruebas porque me preocupa la cobertura del código".

Por último, en esta sección le preguntamos a los profesionales si escriben pruebas porque el proyecto en el que trabajan así lo exige. Comprobamos que el 59,1% se mostraba en desacuerdo con esta pregunta.

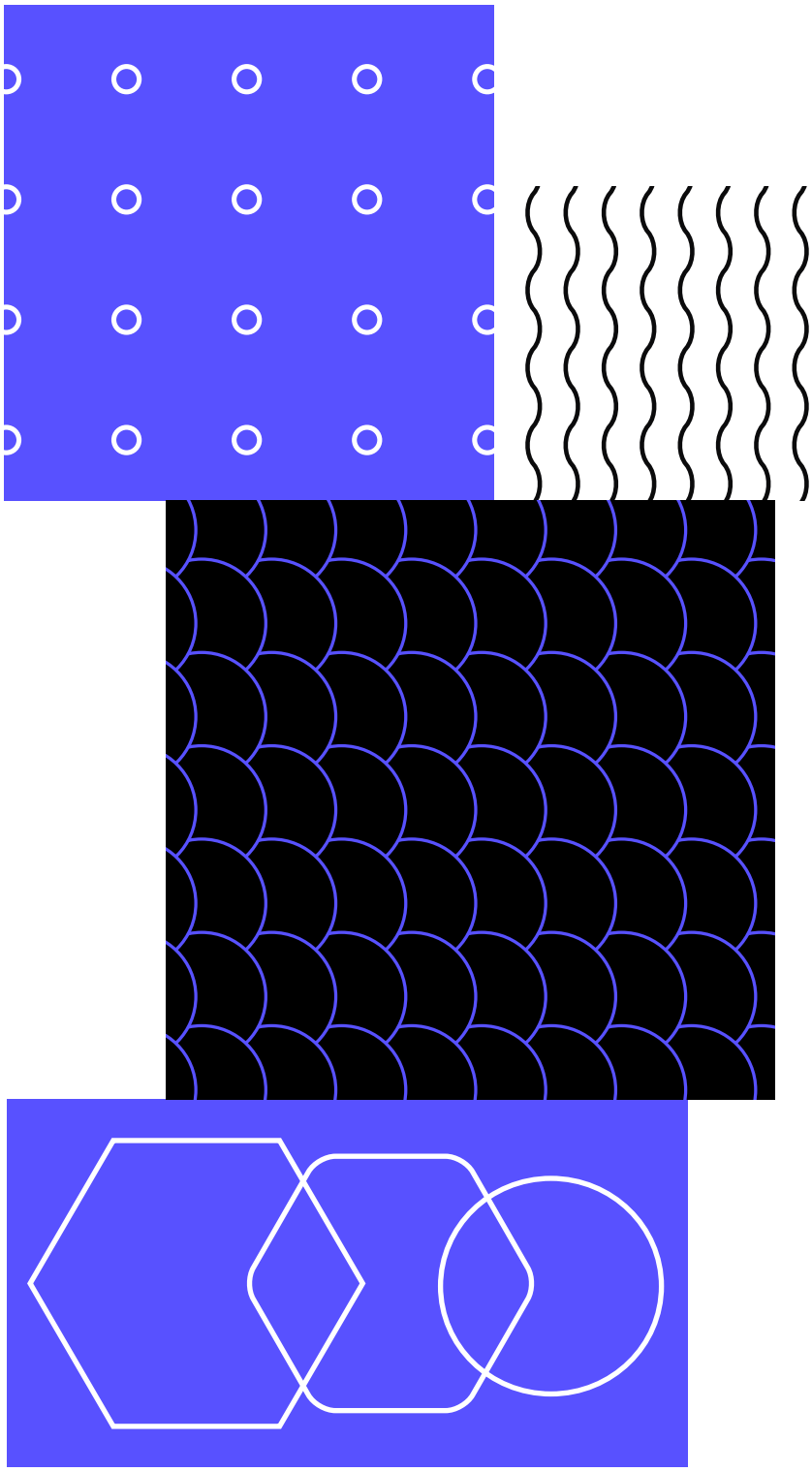
Esto está relacionado con la pregunta "Las empresas en las que trabajo/trabajé exigen TDD como parte de la descripción del puesto". Esto se debe a que las empresas no exigen TDD en sus procesos de selección, como muestra el siguiente gráfico. Evidentemente, a la mayoría de los profesionales no se les exige escribir pruebas para el proyecto en el que trabajan.



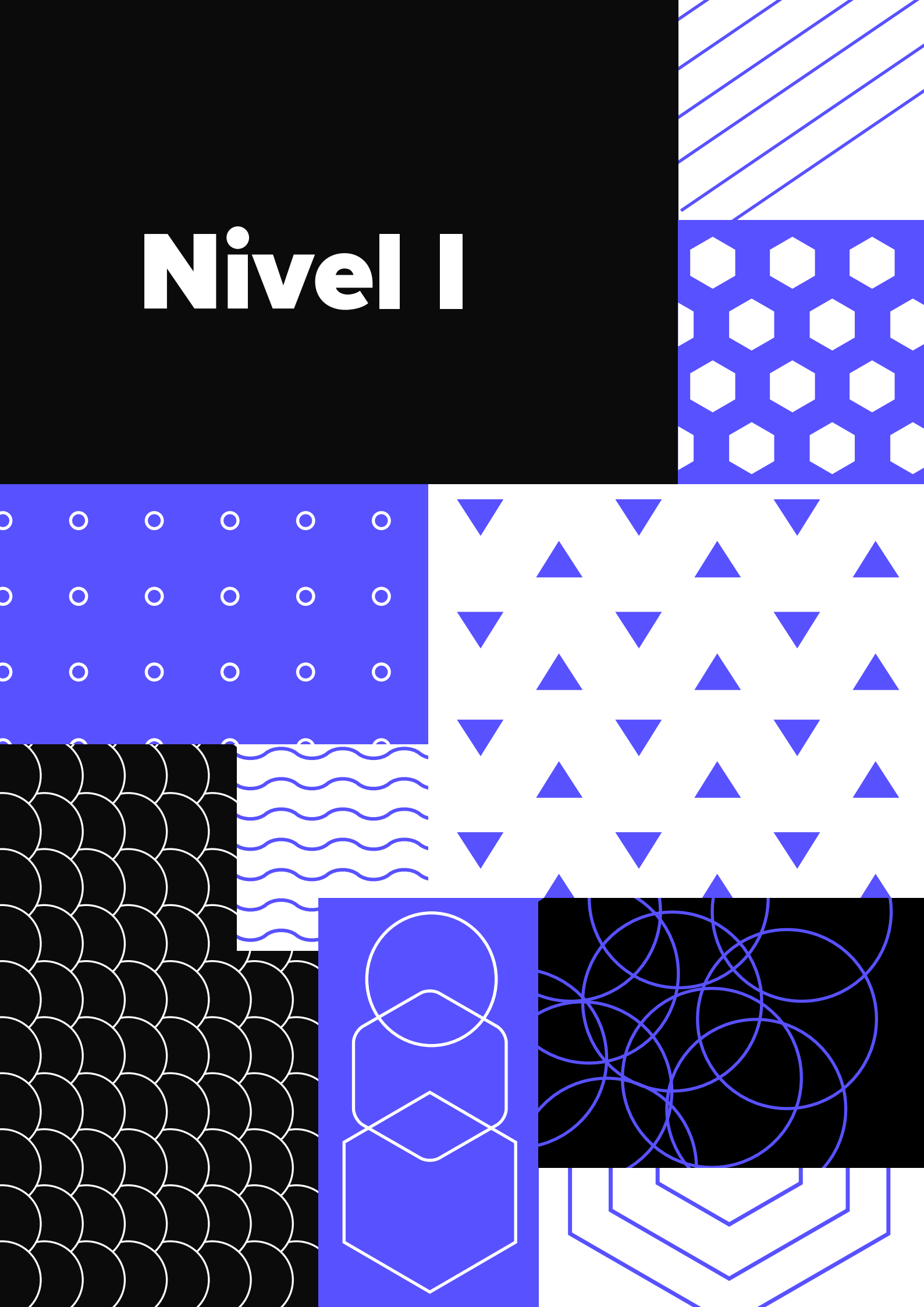
Respuestas a la pregunta: "Escribo pruebas, pero sin TDD porque el proyecto en el que trabajo requiere tests".

A grandes rasgos estos han sido los resultados del sondeo que realizamos sobre el conocimiento de los profesionales respecto a los antipatrones. Además de revelar los resultados obtenidos, nos ha permitido plantearnos algunas cuestiones que requerirán más investigación.

A continuación explicaremos los antipatrones uno por uno y veremos el impacto que cada uno de ellos tiene en los profesionales cuando desarrollan aplicaciones. En el próximo capítulo, repasaremos los antipatrones más comunes para quienes están empezando en TDD.



Nivel I



Nivel I

El primer nivel de esta serie está diseñado para los profesionales que están empezando a testear su código. En este sentido cabe esperar, que esta sea la sección que contenga el mayor número de antipatrones en comparación con el resto. Pensamos que posiblemente esto se deba a la manera en la que los profesionales aprenden TDD (aquí también podemos incluir cualquier enfoque relacionado con escribir el código de pruebas antes del código de producción).

Hemos visto los resultados de la encuesta y una de las claves es que quienes aprenden TDD lo hacen de manera informal.

A lo largo de este nivel, verás temas relacionados con:

- Cómo depender de elementos, tales como el sistema operativo, puede perjudicar la testabilidad.
- Crear dependencias en las que el test se ejecute más allá del sistema operativo también puede perjudicar la testabilidad (por ejemplo, si depende del sistema de archivos).
- Nombrar los casos de prueba sirve para depurar y detectar problemas rápidamente; nombrarlos al azar perjudica su comprensión.
- Agrega nuevos casos de prueba en lugar de contaminar un solo caso de prueba con muchas verificaciones.
- Evita acoplar casos de prueba con el orden en que aparecen en una lista (a menos que el orden tenga un significado).
- Al construir validaciones, en lugar de comparar un objeto entero, enfócate en las propiedades específicas que necesita la prueba.
- Enfócate en el comportamiento deseado y no en las acciones relativamente sencillas, como probar una selección de la base de datos.
- Presta más atención a las pruebas orientadas a async o al tiempo para evitar falsos positivos.
- Saturar el output del test con advertencias o mensajes de error (incluso cuando la prueba está en verde) puede generar confusión; intenta evitarlo en la medida de lo posible.

Sabemos que son muchos temas, así que vamos a tratar de mantener las secciones separadas entre sí para que cada una de ellas pueda consumirse individualmente.

El Operating System Evangelist

Es una prueba unitaria que depende de un sistema operativo específico para poder funcionar. Un buen ejemplo sería un caso de prueba que utiliza la secuencia new line para Windows en una validación, pero que se rompe al ser ejecutado en Linux (Carr 2022).

Discutimos el *Operating System Evangelist* en el Episodio 5 de la serie de vídeos[24] sobre antipatrones de TDD organizada por Codurance.

El proyecto Lutris

El *Operating System Evangelist* está relacionado con el grado de acoplamiento del código de prueba al sistema operativo. El acoplamiento puede darse en diferentes aspectos del código, por ejemplo, si se utiliza una ruta específica que sólo existe en Windows.

Para ilustrar este antipatrón, extrajimos el siguiente snippet del proyecto open source Lutris[25]. El objetivo de Lutris es ejecutar en Linux juegos creados para Windows. De por sí, la premisa del proyecto da algunas limitaciones esperadas en el codebase. El resultado obtenido es el siguiente caso de prueba que lanza un proceso Linux:

```
class LutrisWrapperTestCase(unittest.TestCase):
    def test_excluded_initial_process(self):
        "Test that an excluded process that starts a monitored process works"
        env = os.environ.copy()
        env['PYTHONPATH'] = ':'.join(sys.path)
        # run the lutris-wrapper with a bash subshell. bash is "excluded"
        wrapper_proc = subprocess.Popen(
            [
                sys.executable, lutris_wrapper_bin, 'title', '0', '1', 'bash',
                'bash',
                '-c',
                "echo Hello World; exec 1>&-; while sleep infinity; do true;
                done"
            ],
            stdin=subprocess.DEVNULL, stdout=subprocess.PIPE, env=env,
        )
```

El caso de prueba depende de un shell bash para ser ejecutado y, por ende, fallaría si se intenta ejecutar en un entorno Windows. Esto no quiere decir que esté mal, sino que compromete el enfoque del proyecto por el coste de tener una abstracción encima del sistema operativo subyacente.

Al final, para este escenario en específico, podríamos argumentar que es poco probable que Lutris soporte otro sistema operativo que justifique el coste de mantener una abstracción.

---

24 https://www.codurance.com/es/publications/tdd-and-anti-patterns-chapter-5

25 https://github.com/lutris/lutris/blob/f5e8e007b3e492be1fd07ca695ad6e0e25fab1d5/tests/test_lutris_wrapper.py

En el tercer capítulo[26] de su libro “Cosmic Python”, Harry Percival y Bob Gregory compartieron la idea detrás del acoplamiento y la abstracción. En el, discutieron el concepto de usar un filesystem. Específicamente, hablaron sobre la ruta del archivo y las implicaciones de usar una ruta directamente sin tener en cuenta que cualquier abstracción puede llevar a un código acoplado y perjudicar su testabilidad.[27]

El Operating System Evangelist también apareció en el lenguaje de programación Go lang. Específicamente, surgió en un problema que intentaba mitigar las diferencias de los new line characters entre los sistemas operativos Linux y Windows. Este problema forma parte de la definición de este antipatrón: “Un buen ejemplo sería un caso de prueba que utiliza la secuencia newline para Windows en una validación, y que se rompería al ser ejecutado en Linux”. Dentro de ese hilo de problemas de Github, un usuario compartió los problemas que enfrentó cuando necesitaba ejecutar las mismas pruebas en Windows. Afirmó que la mayoría de los errores se deben a la diferencia entre el código salto de línea (LF).

Otro antipatrón relacionado al *Operating System Evangelist* es el *Local Hero*. El *Local Hero* es conocido por tener todo listo localmente para ejecutar una aplicación, pero en cuanto intentas ejecutarla en otra máquina, falla.

Hablaremos del *Local Hero* más adelante, pero para demostrar cómo están conectados, aquí va un ejemplo del código fuente Jenkins:

```
1  @Test
2  public void testWithoutSOptionAndWithoutJENKINS_URL() throws Exception {
3      Assume.assumeThat(System.getenv("JENKINS_URL"), is(nullValue()));
4      // TODO instead remove it from the process env?
5      assertNotEquals(0, launch("java",
6              "-Duser.home=" + home,
7              "-jar", jar.getAbsolutePath(),
8              "who-am-i")
```

26 Capítulo 3, Un breve interludio sobre acoplamiento y abstracciones.

27 El ejemplo utilizado también puede asociarse al patrón de diseño strategy.

```
9       );
10  }
```

Este snippet es particularmente interesante porque quien lo escribió ya había notado que algo olía mal[28] cuando escribió el comentario: '*TODO* instead remove it from the process env?'

Curiosamente, en el libro *The Programmer's Brain*, Felienne Hermans (2021) compartió que añadir notas *TODO* mientras se codifica le recuerda a los programadores de regresar a arreglar el problema. Esto es lo que ella llamó una técnica para ayudar a la memoria prospectiva. Sin embargo, la propia autora destaca que este no suele ser el caso. Este tipo de comentario tiende a quedarse sin resolver en un codebase durante mucho tiempo.[29]

Por último, las katas suelen ser un buen lugar para detectar este tipo de patrones desde el inicio e impulsar una abstracción durante la fase de refactorización.

Por ejemplo, el WordWrap[30] es una kata que divide en new lines el contenido que es más largo de lo esperado. Para una explicación de las diferencias entre los saltos de línea y los sistemas operativos, consulta el post de Baeldung.com.[31]

Aspectos a tener en cuenta

1. Evita depender del sistema operativo; mejor añade una abstracción basada en el contexto siempre que sea posible.

El Local Hero

Es un caso de prueba que, para poder ejecutarse, depende de un aspecto específico del entorno de desarrollo en el que se escribió. Como resultado, el test pasa en su entorno de desarrollo original, pero falla cuando alguien intenta ejecutarlo en otro lugar (Carr 2022).

Discutimos el *Local Hero* en el Episodio 2 de la serie de vídeos[32] sobre antipatrones de TDD organizada por Codurance.

---

28 Los code smells se definen por la sensación que tienen los desarrolladores al leer/escribir código de que algo no es correcto.

29 Así que, la próxima vez que estés en esta situación, piénsatelo dos veces antes de añadir la etiqueta TODO en el código.

30 https://codingdojo.org/kata/WordWrap

31 https://www.baeldung.com/java-string-newline

32 https://www.codurance.com/es/publications/tdd-y-anti-patrones-cap%C3%ADtulo-2

Los antipatrones de TDD preceden al reciente auge del uso de contenedores en el desarrollo de software. Antes, era habitual que existieran diferencias entre la máquina en la que trabajaba el desarrollador y el servidor en el que se ejecutaría la aplicación. A menudo, estos entornos no eran los mismos: la configuración específica de la máquina del desarrollador se interponía durante el proceso de despliegue que llegaba al servidor de producción y, como resultado, bloqueaba el sistema.

Por ejemplo, PHP depende en gran medida de extensiones que pueden o no ser activadas en el servidor. Estas extensiones incluyen hilos, controladores para conectarse a una base de datos, y muchos más.

En este caso, si la persona que desarrolla software dependiera de una versión específica para una extensión determinada, el test se ejecutaría correctamente. Sin embargo, apenas se intentase ejecutar la suite en otra máquina (como un servidor de Integración Continua), fallaría.

No sólo eso, sino que las variables de entorno también pueden entorpecer las pruebas. Por ejemplo, el siguiente código muestra un componente que necesita un URL para cargar una encuesta[33]:

```
1  import { Component } from 'react';
2  import Button from '../../buttons/primary/Primary';
3
4  import '../../../../scss/shake-horizontal.scss';
5  import './survey.scss';
6
7  const config = {
8    surveyUrl: process.env.REACT_APP_SURVEY_URL || '',
9  }
10
11 const survey = config.surveyUrl;
12
13 const mapStateToProps = state => ({
14   user: state.userReducer.user,
15 });
16
17 export class Survey extends Component {
18   /* skipped code */
19
20   componentDidMount = () => { /* skipped code */}
```

33 Parte del código se ha eliminado/modificado intencionadamente para que se adapte al ejemplo. Para obtener más información, sigue el enlace de GitHub.

```
21
22   onSurveyLoaded = () => { /* skipped code */}
23
24   skipSurvey = () => { /* skipped code */}
25
26   render() {
27     if (this.props.user.uid && survey) {
28       return (
29         <div className={`w-full ${this.props.className}`}>
30           {
31             this.state.loading &&
32             <div className="flex justify-center items-center text-white">
33               <h1>Carregando questionario</h1>
34             </div>
35           }
36
37           <iframe
38             src={this.state.surveyUrl}
39             title="survey form"
40             onLoad={this.onSurveyLoaded}
41           />
42
43           {
44             !this.state.loading && this.props.skip &&
45             <Button
46               className="block mt-5 m-auto"
47               description={this.state.buttonDescription}
48               onClick={this.skipSurvey}
49             />
50           }
51         </div>
52       );
53     }
54
55     return (
56       <div className="flex justify-center items-center text-white">
57         <h1 className="shake-horizontal">Ocorreu um erro ao carregar o
           questionario</h1>
58       </div>
59     );
60   }
61 }
```

```
62 /* skipped code */
```

Y aquí está el caso de prueba para estos componentes:

```
1  import { mount } from 'enzyme';
2  import { Survey } from './Survey';
3  import { auth } from '../../../../pages/login/Auth';
4  import Button from '../../buttons/primary/Primary';
5
6  describe('Survey page', () => {
7
8    test('should show up message when survey url is not defined',() => {
9      const wrapper = mount(<Survey user={{}}/>);
10     const text = wrapper.find('h1').text();
11   });
12
13   test('should not load survey when user id is missing', () => {
14     const wrapper = mount(<Survey user={{}} />);
15     const text = wrapper.find('h1').text();
16   });
17
18   test('load survey passing user id as a parameter in the query string', () => {
19     const user = { uid: 'uhiuqwqw-k-woqk-wq--qw' };
20
21     const wrapper = mount(<Survey user={user} />);
22     const url = wrapper.find('iframe').prop('src');
23     expect(url.includes(auth.user.uid)).toBe(true);
24   });
25
26   test('should not up button when it is loading', () => {
27     const user = { uid: 'uhiuqwqw-k-woqk-wq--qw' };
28
29     const wrapper = mount(<Survey user={user} />);
30     expect(wrapper.find(Button).length).toBe(0);
31   });
32
33   test('should not up button when skip prop is not set', () => {
34     const user = { uid: 'uhiuqwqw-k-woqk-wq--qw' };
35
36     const wrapper = mount(<Survey user={user} />);
37     expect(wrapper.find(Button).length).toBe(0);
38   });
```

```
39
40   test('show up button when loading is done and skip prop is true', () => {
41     const user = { uid: 'uhiuqwqw-k-woqk-wq--qw' };
42
43     const wrapper = mount(<Survey user={user} skip={true} />);
44     wrapper.setState({
45       loading: false
46     });
47     expect(wrapper.find(Button).length).toBe(1);
48   });
49 });
```

A pesar de la edad del código (componentes de clase de larga duración en reactjs), cumple bien con su función. Al derivar el comportamiento de los casos de prueba, entendemos que se está produciendo alguna carga basada en la URL de la encuesta y el ID de usuario. Desafortunadamente, los detalles de implementación son los más importantes, y si ejecutamos el caso de prueba para la implementación actual, fallará

```
1  Test Suites: 1 failed, 62 passed, 63 total
2  Tests:       3 failed, 593 passed, 596 total
```

La solución para tal ejecución es exportar una variable de entorno llamada REACT_APP_SURVEY_URL. Bueno, la solución fácil sería utilizar la variable env. pero, la solución a largo plazo consiste en evitar depender de la definición externa y asumir algunos valores por defecto. Aquí están algunas ideas que se me ocurren para arreglar este aspecto correctamente:

- Asume una variable dummy por defecto.
- No utilices cualquier URL y construye las pruebas en base a si lo tienes o no – si no, simplemente omite la ejecución.

Otro ejemplo sería depender del sistema de archivos subyacente. Este problema también se discute en un hilo de Stack Overflow. El problema de un test dependiente es que sólo se ejecutaría en una máquina Windows. Lo ideal sería evitar las dependencias externas utilizando test doubles.

Aspectos a tener en cuenta

1. Sistema de archivos
2. Dependencias en el sistema operativo
3. Gestión de la configuración externa

El Enumerator

Un test unitario que lleva el nombre de cada método de prueba (es decir: prueba1, prueba2, prueba3) es simplemente una enumeración. Cuando se hace esto, se desconoce la intención del caso de prueba y la única forma de conocerla es al leer el código (Carr, 2022).

Discutimos el *Enumerator* en el Episodio 4 de la serie de vídeos[34] sobre antipatrones de TDD organizada por Codurance.

Enumerar los requisitos en una sesión de *brainstorming* puede ser una buena idea. De hecho, crear esta lista puede ser útil para su posterior uso. Por ejemplo, se podría convertir en las nuevas funciones de un proyecto de software.

Como en el software trabajamos con funciones, es buena idea traducirlas al mismo lenguaje y en el mismo orden, de tal modo que verificarlas se convierte en un *checklist*.

Por muy bueno que suene para la organización y el manejo de funciones, traducir tales listas numeradas directamente a código podría generar problemas de legibilidad y para el código de prueba.

Por extraño que parezca, enumerar los casos de prueba es habitual entre personas que son más novatos desarrollando código , por alguna razón, les parece buena idea escribir la misma descripción del test y añadirle un número para identificarla. El siguiente código sirve como ejemplo:

```
1  from status_processor import StatusProcessor
2
3  def test_set_status():
4
5      row_with_status_inactive_1 = dict(
6
7      row_with__status_inactive_2 = dict(
8
9      row_with_status_inactive_3 = dict(
10
11     row_with_status_inactive_3b = dict(
12
13     row_with_status_inactive_4 = dict(
14
15     row_with_status_inactive_5 = dict(
```

34 https://app.livestorm.co/codurance/testing-anti-patterns-episode-4

La pregunta para los nuevos practicantes de codebase es: ¿Qué significa 1? ¿Qué significa 2? ¿Se trata del mismo caso de prueba?. En pocas palabras, el punto clave aquí es dejar espacio para ser explícito sobre lo que se está probando. Esta idea también la explora Martin (2009) en la sección “G25: Reemplaza los números mágicos por constantes con nombre”.

Este primer ejemplo fue en Python, pero este antipatrón también surge en otros lenguajes de programación. El siguiente ejemplo es en Typescript y muestra otro tipo de enumeración de casos de prueba. En este escenario, los nombres de los casos de prueba son de archivos que se utilizan para ejecutar las pruebas.

Ejemplo del Enumerator siendo ejecutado en un canal de GitHub Actions.

Enumerar casos de prueba podría ocultar algunos patrones de negocio que son reemplazados por números, por lo que la intención de lo que se prueba no es visible. Otro problema que surge es de mitigación: si alguna de esas pruebas falla, lo más probable es que el mensaje de error indique un número en lugar de la causa del error.

Aspectos a tener en cuenta

1. ¿Estamos usando 1, 2, 3?
2. ¿La prueba que falló era fácil de entender? y si es así, ¿por qué?

El Free Ride

En lugar de escribir un nuevo método de caso de prueba para probar otra característica o funcionalidad, se puede montar una nueva validación en un caso de prueba existente (Carr 2022).

Discutimos el *Free Ride* en el Episodio 5 de la serie de vídeos[35] sobre antipatrones de TDD organizada por Codurance.

El proyecto Puppeteer

El *Free Ride* es uno de los antipatrones menos populares según las respuestas de nuestro sondeo. Hipótesis que se nos ocurre: quizá se debe a que su nombre dificulta recordar lo que significa.

El *Free Ride* aparece en casos de prueba que normalmente requieren un nuevo caso de prueba para comprobar el comportamiento deseado. Para mantener este suplemento, añadimos una validación y, a veces incluso, lógica dentro del caso de prueba.

Veamos el siguiente ejemplo extraído del proyecto Puppeteer[36]:

```
1  it('Page.Events.RequestFailed', async () => {
2      const { page, server, isChrome } = getTestState();
3
4      await page.setRequestInterception(true);
5      page.on('request', (request) => {
6        if (request.url().endsWith('css')) request.abort();
7        else request.continue();
8      });
9      const failedRequests = [];
10     page.on('requestfailed', (request) => failedRequests.push(request));
11     await page.goto(server.PREFIX + '/one-style.html');
12     expect(failedRequests.length).toBe(1);
13     expect(failedRequests[0].url()).toContain('one-style.css');
14     expect(failedRequests[0].response()).toBe(null);
15     expect(failedRequests[0].resourceType()).toBe('stylesheet');
16
17     if (isChrome)
18       expect(failedRequests[0].failure().errorText).toBe('net::ERR_FAILED');
19     else
```

35 https://www.codurance.com/es/publications/tdd-and-anti-patterns-chapter-5

36 https://github.com/puppeteer/puppeteer/blob/9ca57f190c85c4b6af5665a8cfe4703571e0edde/test/network.spec.ts#L497

```
20        expect(failedRequests[0].failure().errorText).toBe('NS_ERROR_FAILURE');
21      expect(failedRequests[0].frame()).toBeTruthy();
22    });
```

En este ejemplo, el antipatrón *Free Ride* se manifiesta en declaraciones de tipo if/else al final del test. Hay dos casos de prueba en esta única prueba pero, presumiblemente, la idea era reutilizar el código de configuración y agregar una validación adicional dentro del mismo caso de prueba[37].

Una alternativa sería dividir el caso de prueba para centrarnos en un solo escenario a la vez. El Puppeteer ya ha mitigado este problema utilizando una función para manejar este escenario. Si lo utilizamos para dividir los casos de prueba, entonces el primer caso de prueba se centraría en el navegador Chrome:

```
1   itChromeOnly('Page.Events.RequestFailed', async () => {
2       const { page, server } = getTestState();
3
4       await page.setRequestInterception(true);
5       page.on('request', (request) => {
6         if (request.url().endsWith('css')) request.abort();
7         else request.continue();
8       });
9       const failedRequests = [];
10      page.on('requestfailed', (request) => failedRequests.push(request));
11      await page.goto(server.PREFIX + '/one-style.html');
12      expect(failedRequests.length).toBe(1);
13      expect(failedRequests[0].url()).toContain('one-style.css');
14      expect(failedRequests[0].response()).toBe(null);
15      expect(failedRequests[0].resourceType()).toBe('stylesheet');
16      expect(failedRequests[0].failure().errorText).toBe('net::ERR_FAILED');
17      expect(failedRequests[0].frame()).toBeTruthy();
18    });
```

Y luego, el segundo caso se centraría en firefox:

```
1   itFirefoxOnly('Page.Events.RequestFailed', async () => {
2       const { page, server } = getTestState();
3
4       await page.setRequestInterception(true);
5       page.on('request', (request) => {
```

37 Usar la lógica dentro del caso de prueba está relacionado al antipatrón el *Success Against All Odds*. Hablamos de este antipatrón en la sección 9.1 de este E-book.

```
6         if (request.url().endsWith('css')) request.abort();
7         else request.continue();
8       });
9       const failedRequests = [];
10      page.on('requestfailed', (request) => failedRequests.push(request));
11      await page.goto(server.PREFIX + '/one-style.html');
12      expect(failedRequests.length).toBe(1);
13      expect(failedRequests[0].url()).toContain('one-style.css');
14      expect(failedRequests[0].response()).toBe(null);
15      expect(failedRequests[0].resourceType()).toBe('stylesheet');
16      expect(failedRequests[0].failure().errorText).toBe('NS_ERROR_FAILURE');
17      expect(failedRequests[0].frame()).toBeTruthy();
18    });
```

En sí, la lógica dentro de este caso de prueba es una indicación de que el antipatrón *Free Ride* está jugando un papel. El ejemplo de Puppeteer puede ser mejorado incluso más.38

Ahora que hemos dividido la lógica en dos casos de prueba separados, tenemos algunos pedazos de código duplicados (esto podría ser un argumento para adoptar el *Free Ride*). En este caso, el *framework* de pruebas puede ayudarnos.

Para evitar la duplicación de código en este escenario, podríamos utilizar el hook beforeEach y mover la configuración requerida allí.

El proyecto Jenkins

Más allá del proyecto Puppeteer, hay otras maneras en las que el Free Ride puede implementarse. Ahora pasaremos a otro proyecto de código abierto que también demuestra el antipatrón el *Free Ride*.

El siguiente código fue extraído del proyecto Jenkins y también muestra las características distintivas del *Free Ride*. Pero antes de adentrarnos en este punto echemos un vistazo al código fuente:

```
1  public class ToolLocationTest {
2          @Rule
3          public JenkinsRule j = new JenkinsRule();
4
5          @Test
6          public void toolCompatibility() {
```

38 Es importante destacar que el proyecto Puppeteer también acogió el pull request que arregló el *Free Ride* descrito en esta sección. Para más detalles, consulta el siguiente pull request: https://github.com/puppeteer/puppeteer/pull/8095

```
7            Maven.MavenInstallation[] maven = j.jenkins.getDescriptorByType(Maven.
             DescriptorImpl.class).getInstallations();
8            assertEquals(1, maven.length);
9            assertEquals("bar", maven[0].getHome());
10           assertEquals("Maven 1", maven[0].getName());
11
12           Ant.AntInstallation[] ant = j.jenkins.getDescriptorByType(Ant.
             DescriptorImpl.class).getInstallations();
13           assertEquals(1, ant.length);
14           assertEquals("foo", ant[0].getHome());
15           assertEquals("Ant 1", ant[0].getName());
16       JDK[] jdk = j.jenkins.getDescriptorByType(JDK.DescriptorImpl.class).
         getInstallations();
17           assertEquals(Arrays.asList(jdk), j.jenkins.getJDKs());
18           assertEquals(2, jdk.length); // JenkinsRule adds a 'default' JDK
19           assertEquals("default", jdk[1].getName()); // make sure it's really
             that we're seeing
20           assertEquals("FOOBAR", jdk[0].getHome());
21           assertEquals("FOOBAR", jdk[0].getJavaHome());
22           assertEquals("1.6", jdk[0].getName());
23       }
24   }
```

En este caso, otro método para evitar el *Free Ride* sería dividir los casos de prueba:

```
1   public class ToolLocationTest {
2       @Test
3       @LocalData
4       public void shouldBeCompatibleWithMaven() {
5           Maven.MavenInstallation[] maven = j.jenkins.getDescriptorByType(Maven.
            DescriptorImpl.class).getInstallations();
6           assertEquals(1, maven.length);
7           assertEquals("bar", maven[0].getHome());
8           assertEquals("Maven 1", maven[0].getName());
9       }
10      @Test
11      @LocalData
12      public void shouldBeCompatibleWithAnt() {
13          Ant.AntInstallation[] ant = j.jenkins.getDescriptorByType(Ant.
            DescriptorImpl.class).getInstallations();
14          assertEquals(1, ant.length);
```

```
15            assertEquals("foo", ant[0].getHome());
16            assertEquals("Ant 1", ant[0].getName());
17        }
18        @Test
19        @LocalData
20        public void shouldBeCompatibleWithJdk() {
21            JDK[] jdk = j.jenkins.getDescriptorByType(JDK.DescriptorImpl.class).
getInstallations();
22            assertEquals(Arrays.asList(jdk), j.jenkins.getJDKs());
23            assertEquals(2, jdk.length); // JenkinsRule adds a 'default' JDK
24            assertEquals("default", jdk[1].getName()); // make sure it's really
that we're seeing
25            assertEquals("FOOBAR", jdk[0].getHome());
26            assertEquals("FOOBAR", jdk[0].getJavaHome());
27            assertEquals("1.6", jdk[0].getName());
28        }
29    }
```

Esta división también aportaría la ventaja adicional de facilitar en gran medida la identificación de las causas del fallo de una prueba.

Aspectos a tener en cuenta

1. En caso de que una prueba tenga validaciones que afirman comportamientos diferentes lo ideal es dividirla en pruebas separadas.
2. Se puede empezar con todo en un único caso de prueba, pero no refactorizar estas pruebas es algo que hay que vigilar.

El Sequencer

Una prueba unitaria que depende de los elementos de una lista desordenada aparece en ese mismo orden en las validaciones (Carr 2022).

Discutimos el Sequencer en el Episodio 4 de la serie de vídeos[39] sobre antipatrones de TDD organizada por Codurance.

El *Sequencer* pone foco en el tema discutido en el blog 'Testing Assertions' (Marabesi 2022), que describe formas de mejorar el feedback de los casos de prueba en función del tipo de validación utilizada (en este caso, utilizando jest como framework de pruebas). Más concretamente, la sección sobre **Array Containing** describe lo que es el *Sequencer*.

En resumen, el antipatrón el *Sequencer* aparece cuando una lista desordenada

39 https://www.codurance.com/es/publications/tdd-and-anti-patterns-chapter-4

se utiliza para verificar que si se adhiere a un orden dado; en otras palabras, dando la idea de que los elementos de la lista obligatoriamente deben estar ordenados. A menudo, observar que todo funcionaba como se esperaba pero no en el orden esperado es una pérdida de tiempo.

El siguiente ejemplo muestra el *Sequencer* puesta en práctica: el caso de prueba comprueba si la fruta deseada está dentro de la lista. El objetivo aquí es saber si la fruta está o no en la lista independientemente de la posición en la que pueda aparecer:

```
const expectedFruits = ['banana', 'mango', 'watermelon']

expect(expectedFruits[0]).toEqual('banana')
expect(expectedFruits[1]).toEqual('mango')
expect(expectedFruits[0]).toEqual('watermelon')
```

Como no nos importa la posición, el uso del utility arrayContaining podría ser una mejor opción ya que hace que la intención sea explícita para los lectores.

```
const expectedFruits = ['banana', 'mango', 'watermelon']

const actualFruits = () => ['banana', 'mango', 'watermelon']

expect(expectedFruits).toEqual(expect.arrayContaining(actualFruits))
```

Es importante tener en cuenta que *arrayContaining* ignora tanto la posición de los elementos como los elementos extra. Si el código bajo prueba se preocupa del número exacto de elementos, sería mejor utilizar una combinación de validaciones. Este comportamiento se describe en la documentación oficial de Jest.

El ejemplo que utiliza Jest da una pista sobre qué esperar en las codebases que tienen este antipatrón. Aún así, la siguiente ilustración muestra un escenario en el que el sequencer aparece en un archivo CSV.40

```
def test_predictions_returns_a_dataframe_with_automatic_predictions(self,form):
  order_id = "51a64e87-a768-41ed-b6a5-bf0633435e20"
  order_info = pd.DataFrame({"order_id": [order_id], "form": [form],})
  file_path = Path("tests/data/prediction_data.csv")
  service = FileRepository(file_path)

  result = get_predictions(main_service=service, order_info=order_info)
```

40 Agradecemos a Javier Martínez Alcantara por elaborar este ejemplo y compartirlo en el cuarto episodio de la serie de vídeos sobre antipatrones en Codurance.

```
8
9     assert list(result.columns) == [“id”, “quantity”, “country”, “form”, “order_
      id”]
```

En la línea 4, el archivo CSV se carga para ser usado durante el test. A continuación, la variable result es contra la que verificamos y, en la línea 9, tenemos la validación contra la columnas que vienen del archivo.

Los archivos CSV utilizan la primera fila como cabecera del archivo separada por una coma. En la primera fila se define el nombre de las columnas, mientras que las líneas siguientes registran los datos que debe tener cada columna. Si se modifica el orden de columnas (en este caso, cambiando el país y el formulario) en el CSV, veremos el siguiente error:

```
1   tests/test_predictions.py::TestPredictions::test_predictions_returns_a_
dataframe_with_automatic_predictions FAILED [100%]
2   tests/test_predictions.py:16 (TestPredictions.test_predictions_returns_a_
dataframe_with_automatic_predictions)
3   [‘id’, ‘quantity’, ‘form’, ‘country’, ‘order_id’] != [‘id’, ‘quantity’,
‘country’, ‘form’, ‘order_id’]
4
5   Expected :[‘id’, ‘quantity’, ‘country’, ‘form’, ‘order_id’]
6   Actual   :[‘id’, ‘quantity’, ‘form’, ‘country’, ‘order_id’]
```

En este caso de prueba, las columnas existen independientemente del orden. Al final, lo que más importa es tener las columnas con sus respectivos datos independientemente de su orden.

Un mejor enfoque sería sustituir la línea 2 representada anteriormente por la siguiente validación:

```
1   assert set(result.columns) == {“id”, “quantity”, “country”, “form”, “order_id”}
```

El Sequencer es un antipatrón que no se detecta frecuentemente porque es fácil de escribir, y también porque el conjunto de pruebas suele resultar verde. Dicho antipatrón se revela cuando a alguien le cuesta depurar un fallo que supuestamente debió haber superado la prueba.

Aspectos a tener en cuenta

1. Conoce tus estructuras de datos.
2. Piensa en qué papel desempeña el orden en una colección.

El Nitpicker

Es una prueba unitaria que compara un output entero cuando en realidad sólo le interesan pequeñas partes de él. Por este motivo, la prueba se mantiene continuamente en línea con detalles que normalmente carecen de importancia. Endémico en las pruebas de aplicaciones web (Carr 2022).

Discutimos el *Nitpicker* en el Episodio 3 de la serie de vídeos[41] sobre antipatrones de TDD organizada por Codurance.

Por definición, el *Nitpicker* se observa en aplicaciones web en las que la necesidad de verificar el output se centra en un objeto completo en lugar de en la propiedad específica requerida. Como demostramos en el primer ejemplo, esto es común para las estructuras JSON.

Laravel Assertions

El siguiente código afirma que una aplicación ha sido eliminada. En este contexto, una aplicación es una entrada normal en la base de datos con la etiqueta "aplicación."

Ten en cuenta que este ejemplo en PHP se utiliza para verificar el output exacto de la solicitud HTTP. Ni más ni menos.

```php
<?php
public function testDeleteApplication()
{
    $response = $this->postApplication();

    $this->assertFalse($response->error);

    $this->delete('api/application/' . $response->data)
        ->assertExactJson([                    // is this needed?
        'data' => (string) $response->data,
        'error' => false
    ]);
}
```

Esta prueba es frágil por un motivo específico: si añadimos otra propiedad a la respuesta, fallará reclamando que el JSON ha cambiado. Dicho fallo sería útil para eliminar esas propiedades; sin embargo, sería poco funcional para añadir una propiedad nueva.

41 https://www.codurance.com/es/publications/tdd-y-anti-patrones-capitulo-3

La “solución” consistiría en sustituir la idea “exacta” en esta afirmación por una menos estricta, como la siguiente:

```php
<?php
public function testDeleteApplication()
{
    $response = $this->postApplication();

    $this->assertFalse($response->error);

    $this->delete('api/application/' . $response->data)
        ->assertJson([                          // <!--- changing this assertion
        'data' => (string) $response->data,
        'error' => false
    ]);
}
```

Este cambio afirma que el fragmento deseado está efectivamente en el output. No importa si hay otras propiedades en el output, siempre y cuando la deseada esté ahí. Este simple cambio permite alejarnos de la fragilidad que originalmente caracterizaba a la prueba.

El proyecto AWS CloudFront URL Signature Utility

Otra forma de no afrontar el *Nitpicker* es buscando sólo las propiedades que te preocupan. El siguiente código es de un proyecto open-source que gestiona el proceso de firma para acceder a un recurso en Amazon S3[42]:

```
[caption=]
describe('#getSignedCookies()', function() {
  it('should create cookies object', function(done) {
    var result = CloudfrontUtil.getSignedCookies(
      'http://foo.com', defaultParams);

    expect(result).to.have.property('CloudFront-Policy');
    expect(result).to.have.property('CloudFront-Signature');
    expect(result).to.have.property('CloudFront-Key-Pair-Id');
    done();
  });
});
```

42 Accede al código fuente en este enlace: https://github.com/jasonsims/aws-cloudfront-sign/blob/master/test/lib/cloudfrontUtil.test.js#L235

En lugar de comprobar todas las propiedades a la vez, el código cuenta con tres validaciones que verifican si efectivamente tiene la propiedad deseada, independientemente del output.

El proyecto Metrik

Otro ejemplo de cómo abordar dicha validación es con el código extraído de un proyecto open-source que pretende recopilar y procesar las Cuatro Métricas Clave (Radziwill 2020) denominado *Metrik*:[43]

```
1  @Test
2  fun `should calculate CFR correctly by monthly and the time split works well (
cross a calendar month)`() {
3       val requestBody = """ { skipped code } """.trimIndent()
4       RestAssured
5           .given()
6           .contentType(ContentType.JSON)
7           .body(requestBody)
8           .post("/api/pipeline/metrics")
9           .then()
10          .statusCode(200)
11          .body("changeFailureRate.summary.value", equalTo(30.0F))
12          .body("changeFailureRate.summary.level", equalTo("MEDIUM"))
13          .body("changeFailureRate.details[0].value", equalTo("NaN"))
14          .body("changeFailureRate.details[1].value", equalTo("NaN"))
15          .body("changeFailureRate.details[2].value", equalTo(30.0F))
16 }
```

En lugar de utilizar todo el objeto como comparación como vimos en el primer ejemplo, el framework RestAssured[44] se utiliza una vez más para buscar propiedades individuales dentro del output.

Los frameworks de pruebas suelen ofrecer este tipo de utilidades para ayudar a los profesionales a probar su código de esta manera. En el primer ejemplo, el framework PHP Laravel utiliza la sintaxis assertJson/assertExactJson.[45]

El segundo ejemplo utiliza la biblioteca de pruebas Chai para demostrar cómo verificar propiedades específicas dentro de un objeto.

Por último, RestAssured es la biblioteca utilizada para mostrar cómo trabajar con

---

43 https://github.com/thoughtworks/metrik

44 https://rest-assured.io

45 https://laravel.com/docs/9.x/http-tests#verifying-exact-match

el *Nitpicker* dentro del ecosistema Kotlin.

Aspectos a tener en cuenta

1. Valida sólo contra las propiedades y valores que te interesen.
2. Puede generalizarse a otras aplicaciones (por ejemplo, CLI)

El Dodger

Consiste en una prueba unitaria con muchos tests para efectos secundarios menores (y presumiblemente fáciles de testear) que nunca comprueba el comportamiento central deseado. A veces podemos encontrarlo en los tests relacionados con el acceso a bases de datos, en los cuales se solicita un método, y la prueba selecciona y ejecuta en la base de datos validaciones contra el resultado (Carr 2022).

Discutimos el *Dodger* en el Episodio 3 de la serie de vídeos[46] sobre antipatrones de TDD organizada por Codurance.

El *Dodger* es el antipatrón más común cuando se empieza a desarrollar software con el enfoque de escribir pruebas primero. Antes de ver el ejemplo de un código, vamos a explicar un poco más por qué surge este antipatrón.

Desarrollar código siguiendo las prácticas de TDD implica escribir primero el test del mismo. La regla es: empezar con un test que falla, hacer que pase, y luego refactorizar el diseño. Por muy sencillo que sea, cuando seguimos este proceso surgen momentos en los que nos preguntamos: “¿qué debería probar?”.

Lo habitual es empezar a escribir pruebas para una clase y una clase de producción, lo que significa que tendrían una relación de 1-1. Luego surge la siguiente pregunta: “¿Cómo de pequeño debe ser el ámbito de las pruebas?”. Como el concepto “pequeño” depende del contexto, no es obvio cuál es el ámbito más pequeño aceptable para una prueba.

Estas dos preguntas son muy comunes cuando se empieza a practicar TDD, y podrían llevar al antipatrón *Dodger*. Este último consiste en testear el código de implementación específico en lugar del comportamiento deseado[47]. Para representarlo, presentamos el siguiente código de producción:

---

46 https://www.codurance.com/es/publications/tdd-y-anti-patrones-capitulo-3

47 En esta charla (https://www.youtube.com/watch?v=APFbb5MwLEU) Mario Cervera explica qué es el comportamiento y cómo se aplica al Test Driven Development (TDD).

```php
<?php

namespace Drupal\druki_author\Data;

use Drupal\Component\Utility\UrlHelper;
use Drupal\Core\Language\LanguageManager;
use Drupal\Core\Locale\CountryManager;

/**
 * Provides author value object.
 */
final class Author {

  /** skipped protected properties to fit code here  */

  /**
   * Builds an instance from an array.
   *
   * @param string $id
   *   The author ID.
   * @param array $values
   *   The author information.
   */
  public static function createFromArray(string $id, array $values): self {
    $instance = new self();
    if (!\preg_match('/^[a-zA-Z0-9_-]{1,64}$/', $id)) {
      throw new \InvalidArgumentException('Author ID contains not allowed characters, please fix it.');
    }
    $instance->id = $id;

    if (!isset($values['name']) || !\is_array($values['name'])) {
      throw new \InvalidArgumentException("The 'name' value is missing or incorrect.");
    }
    if (\array_diff(['given', 'family'], \array_keys($values['name']))) {
      throw new \InvalidArgumentException("Author name should contains 'given' and 'family' values.");
    }
    $instance->nameGiven = $values['name']['given'];
    $instance->nameFamily = $values['name']['family'];

```

```
40    if (!isset($values['country'])) {
41      throw new \InvalidArgumentException("Missing required value 'country'.");
42    }
43    $country_list = \array_keys(CountryManager::getStandardList());
44    if (!\in_array($values['country'], $country_list)) {
45      throw new \InvalidArgumentException('Country value is incorrect. It should
      be valid ISO 3166-1 alpha-2 value.');
46    }
47    $instance->country = $values['country'];
48
49    if (isset($values['org'])) {
50      if (!\is_array($values['org'])) {
51        throw new \InvalidArgumentException('Organization value should be an
        array.');
52      }
53      if (\array_diff(['name', 'unit'], \array_keys($values['org']))) {
54        throw new \InvalidArgumentException("Organization should contains 'name'
        and 'unit' values.");
55      }
56      $instance->orgName = $values['org']['name'];
57      $instance->orgUnit = $values['org']['unit'];
58    }
59
60    if (isset($values['homepage'])) {
61      if (!UrlHelper::isValid($values['homepage']) ||
      !UrlHelper::isExternal($values['homepage'])) {
62        throw new \InvalidArgumentException('Homepage must be valid external
        URL.');
63      }
64      $instance->homepage = $values['homepage'];
65    }
66
67    if (isset($values['description'])) {
68      if (!\is_array($values['description'])) {
69        throw new \InvalidArgumentException('The description should be an array
        with descriptions keyed by a language code.');
70      }
71      $allowed_languages = \array_
      keys(LanguageManager::getStandardLanguageList());
72      $provided_languages = \array_keys($values['description']);
73      if (\array_diff($provided_languages, $allowed_languages)) {
74        throw new \InvalidArgumentException('The descriptions should be keyed by
```

```
          a valid language code.');
75      }
76      foreach ($values['description'] as $langcode => $description) {
77        if (!\is_string($description)) {
78          throw new \InvalidArgumentException('Description should be a
            string.');
79        }
80        $instance->description[$langcode] = $description;
81      }
82    }
83
84    if (isset($values['image'])) {
85      if (!\file_exists($values['image'])) {
86        throw new \InvalidArgumentException('The image URI is incorrect.');
87      }
88      $instance->image = $values['image'];
89    }
90
91    if (isset($values['identification'])) {
92      if (isset($values['identification']['email'])) {
93        if (!\is_array($values['identification']['email'])) {
94          throw new \InvalidArgumentException('Identification email should be an
            array.');
95        }
96        $instance->identification['email'] = $values['identification']['email'];
97      }
98    }
99
100   return $instance;
101 }
102
103 public function getId(): string {
104   return $this->id;
105 }
106
107 public function getNameFamily(): string {
108   return $this->nameFamily;
109 }
110
111 public function getNameGiven(): string {
112   return $this->nameGiven;
113 }
```

```
114
115   public function getCountry(): string {
116     return $this->country;
117   }
118
119   public function getOrgName(): ?string {
120     return $this->orgName;
121   }
122
123   public function getOrgUnit(): ?string {
124     return $this->orgUnit;
125   }
126
127   public function getHomepage(): ?string {
128     return $this->homepage;
129   }
130
131   public function getDescription(): array {
132     return $this->description;
133   }
134
135   public function getImage(): ?string {
136     return $this->image;
137   }
138
139   public function checksum(): string {
140     return \md5(\serialize($this));
141   }
142
143   public function getIdentification(?string $type = NULL): array {
144     if ($type) {
145       if (!isset($this->identification[$type])) {
146         return [];
147       }
148       return $this->identification[$type];
149     }
150     return $this->identification;
151   }
152 }
```

El objetivo es validar el objeto Author antes de crearlo a partir de un array. Para ser creado, el array dado debe contener datos válidos; de lo contrario, surgirá

una excepción. A continuación, el código de prueba:

```php
<?php
/**
 * Tests that objects works as expected.
 */
public function testObject(): void {
  $author = Author::createFromArray($this->getSampleId(), $this->getSampleValues());
  $this->assertEquals($this->getSampleId(), $author->getId());
  $this->assertEquals($this->getSampleValues()['name']['given'], $author->getNameGiven());
  $this->assertEquals($this->getSampleValues()['name']['family'], $author->getNameFamily());
  $this->assertEquals($this->getSampleValues()['country'], $author->getCountry());
  $this->assertEquals($this->getSampleValues()['org']['name'], $author->getOrgName());
  $this->assertEquals($this->getSampleValues()['org']['unit'], $author->getOrgUnit());
  $this->assertEquals($this->getSampleValues()['homepage'], $author->getHomepage());
  $this->assertEquals($this->getSampleValues()['description'], $author->getDescription());
  $this->assertEquals($this->getSampleValues()['image'], $author->getImage());
  $this->assertEquals($this->getSampleValues()['identification'], $author->getIdentification());
  $this->assertEquals($this->getSampleValues()['identification']['email'], $author->getIdentification('email'));
  $this->assertEquals([], $author->getIdentification('not exist'));
  $this->assertEquals($author->checksum(), $author->checksum());
}
```

Lo primero que se nota al revisar el código es que para cambiar la manera de obtener el nombre del autor (por ejemplo al renombrar el método) también es necesario cambiar el código de prueba. A pesar de que el comportamiento deseado no haya cambiado, la validación (el comportamiento actual) sigue siendo necesaria.

Un enfoque alternativo consistiría en dividir el caso de prueba único en múltiples casos de prueba. De este modo, capturamos la excepción deseada si pasa un valor no deseado, y lo encapsulamos en una clase validador para evitar el acoplamiento del código de prueba y de producción.

Aspectos a tener en cuenta

1. La relación entre la clase de prueba y la clase de producción es de 1-1.
2. En lugar de pensar en los detalles específicos de la implementación, dedícate a testear el comportamiento.

El Liar

Consiste en una prueba unitaria completa que supera todos los casos de prueba que tiene y parece válida, pero que, tras una inspección más minuciosa, se descubre que en realidad no prueba en absoluto el objetivo previsto (Carr 2022).

Discutimos el *Liar* en el Episodio 1 de la serie de vídeos[48] sobre antipatrones de TDD organizada por Codurance.

El *Liar* es uno de los antipatrones más comunes debido a que tiende a esconderse en el código fuente. Sólo se revela haciendo una inspección más detallada. Existen al menos dos razones para detectar este tipo de problemas en las codebases:

1. Casos de prueba asíncronos.
2. Casos de prueba orientadas al tiempo.

La primera está bien explicada en la documentación oficial de Jest[49]. Probar código asíncrono se vuelve complicado ya que se basa en un valor futuro que puedes o no recibir (jestjs.io 2021). El siguiente código es un ejemplo reproducido de la documentación oficial de Jest (la documentación indica en un comentario de código que no se use el siguiente código de prueba en proyectos reales).

Async Test with Jest

```
// Don't do this!
test('the data is peanut butter', () => {
  function callback(data) {
    expect(data).toBe('peanut butter');
  }

  fetchData(callback);
});
```

Volviendo al antipatrón el *Liar*, esta prueba pasaría sin problemas; aunque, debido a cómo está escrita, en realidad sería una mentira. El enfoque correcto es esperar a que la función asíncrona termine su ejecución y darle a Jest el control

48 https://www.codurance.com/es/publications/tdd-anti-patterns-chapter-1

49 El ejemplo utilizado aquí es de Jest, pero también puede ser encontrado en otros frameworks de pruebas.

sobre la ejecución del flujo de nuevo.

```
test('the data is peanut butter', done => {
  function callback(data) {
    try {
      expect(data).toBe('peanut butter');
      done(); // invokes jest flow again, saying: "look I am ready now!"
    } catch (error) {
      done(error);
    }
  }

  fetchData(callback);
});
```

Respecto a los casos de prueba orientados al tiempo, Martin Fowler explica las razones por las que se da este caso (Fowler 2011), y aquí compartiremos algunas opiniones que coinciden con lo que él escribió.

El 'asíncrono' es una fuente de no-determinismo y, como ya vimos en el ejemplo anterior de Jest, debemos tener cuidado con eso. Adicionalmente, los hilos en el código de prueba también merecen un cuidado especial. Si necesitas manejarlos, asegúrate de que funcionen como se espera.

Por otra parte, si no se manejan adecuadamente, las pruebas orientadas al tiempo a veces pueden fallar sin motivo aparente. Por lo tanto, los profesionales tienden a adoptar test doubles para manejar las fechas de una manera que puedan controlarlas sin que se acoplen a un tiempo real. Así se evita la situación en la que el día en que se escribe el código, la prueba pasa, pero al día siguiente falla.

Aspectos a tener en cuenta

3. Las pruebas asíncronas pueden hacer que los tests lleguen a conclusiones erróneas. Ten cuidado con ciertos ejecutores de pruebas.

El Loudmouth

Es una prueba unitaria (o conjunto de pruebas) que llena la consola con mensajes de diagnóstico, mensajes de registro y otros parloteos varios; incluso cuando los tests están en verde. A veces, durante la creación de la prueba, se desea ver manualmente el output y, a pesar de que no sea necesario, se deja en el código. (Carr 2022).

Discutimos el *Loudmouth* en el Episodio 3 de la serie de vídeos[50] sobre antipatrones de TDD organizada por Codurance.

Al desarrollar, es habitual añadir algunos rastros temporales al código para ayudar al desarrollador a confirmar si el código se comporta o no como se espera. Este proceso suele denominarse depuración (o *debugging* en inglés), y se utiliza a menudo cuando los desarrolladores necesitan aclarar su entendimiento de un fragmento del código.

Las personas que practican TDD argumentan que una vez que se practica TDD, no se necesita ninguna herramienta de depuración (Freeman y Pryce 2009), ya sea una print statement o añadir puntos de interrupción en el código.

Pero, ¿qué ocurre si no tienes tanta experiencia con TDD?

El proyecto testeable

A menudo la respuesta está en depurar el código y usar las pruebas de guía. Por ejemplo, el siguiente código muestra un código de prueba que puede encargarse de un error si recibe una pieza inválida de código JavaScript. Ten en cuenta que el código se utiliza para analizar el código JavaScript y actuar sobre su resultado:

```
1  test.each([['function']])(
2    'should not bubble up the error when an invalid source code is provided,
3    (code) => {
4      const strategy = jest.fn();
5
6      const result = Reason(code, strategy);
7      expect(strategy).toHaveBeenCalledTimes(0);
8      expect(result).toBeFalsy();
9    }
10 );
```

La verificación es sencilla. Esta asegura que la estrategia deseada no sea llamada ya que el código es una pieza inválida de código JavaScript. También comprueba si el resultado es un valor booleano falso. Ahora veamos cómo es la implementación de esta prueba:

```
1  const Reason = function(code, strategy) {
2    try {
3      const ast = esprima.parseScript(code);
4
```

50 https://www.codurance.com/es/publications/tdd-y-anti-patrones-capitulo-3

```
    if (ast.body.length > 0) {
      return strategy(ast);
    }
  } catch (error) {
    console.warn(error);           // < ---------- this is loud
    return false;
  }
};
```

Idealmente, al trabajar de forma TDD, la declaración de console.log utilizada sería mocked desde el principio. Esto se debe a que requiere la verificación de cuándo fue llamada y con qué mensaje. Este primer indicio apunta a un enfoque que no testea primero. La siguiente imagen muestra lo que genera el antipatrón *Loudmouth*. Aunque las pruebas están en verde, hay un mensaje de advertencia: ¿El código pasó la prueba? ¿El cambio rompió algo?

(Freeman y Pryce 2009) dan una idea de por qué el logging (como es el caso de este console.log) debe ser tratado como una función y no como un registro aleatorio utilizado por cualquier razón.

El siguiente snippet muestra una posible implementación que retira console.log y evita que el mensaje se muestre durante la ejecución de la prueba:

```
const originalConsole = globalThis.console;

beforeEach(() => {
  globalThis.console = {
    warn: jest.fn(),
    error: jest.fn(),
    log: jest.fn()
  };
});

afterEach(() => {
  globalThis.console = originalConsole;
});
```

Con la consola extraída, ahora es posible validar su uso en lugar de imprimir el output mientras se ejecutan las pruebas. La versión sin el *Loudmouth* sería la siguiente:

```
const originalConsole = globalThis.console;

beforeEach(() => {
  globalThis.console = {
    warn: jest.fn(),
    error: jest.fn(),
    log: jest.fn()
  };
});

afterEach(() => {
    globalThis.console = originalConsole;
});

test.each([['function']])(
  'should not bubble up the error when an invalid source code is provided',
  (code) => {
    const strategy = jest.fn();

    const result = Reason(code, strategy);
    expect(strategy).toHaveBeenCalledTimes(0);
    expect(result).toBeFalsy();
    expect(globalThis.console.warn).toHaveBeenCalled(); // < -- did it warn?
  }
);

```

El antipatrón *Loudmouth* puede potencialmente hacer que el desarrollador dude si la prueba que está pasando está pasando por la razón correcta. Esto se debe a que el output del logging adicional contamina el output de las pruebas mientras estas se ejecutan.

Aspectos a tener en cuenta

1. ¡Límpia!
2. Trata los registros como una función, y testealos.

Nivel II

Nivel II

El nivel II se dirige a aquellos principiantes que se encuentran en un nivel intermedio. A medida que avanzamos en el enfoque dirigido por pruebas, las cosas empiezan a complicarse ligeramente. Ya sabemos qué probar y puede que también nos sintamos cómodos configurando cualquier tipo de entorno para las pruebas pero, debido a la falta de confianza, se pierden algunos principios.

Como ya tenemos cierta experiencia, es posible que veamos casos de prueba que deberían estar en rojo, pero cuyo resultado es verde.

Algunos temas a tener en cuenta cuando estés en este nivel:

- Evitar escribir un test que pase primero.
- Evitar indagar en otras implementaciones de objetos para configurar un caso de prueba.
- Cuando una prueba falla y es difícil detectar la raíz del problema, es posible que haya una dependencia oculta.
- Evitar capturar excepciones sólo para hacer pasar una prueba.
- Evitar compartir estado entre pruebas siempre que sea posible.
- Evitar depender de las excepciones para que la prueba pase. En su lugar, hay que hacer que las validaciones sean explícitas.

El Success Against All Odds

Es una prueba escrita que pasa primero en lugar de fallar primero. Como consecuencia, el caso de prueba siempre pasa aun cuando debe fallar (Carr 2022).

Discutimos el *Success Against All Odds* en el Episodio 5 de la serie de videos[51] sobre antipatrones de TDD organizada por Codurance.

El *Success Against All Odds* es un antipatrón asociado a la falta de un enfoque de test-first. En este caso, el practicante sigue la prueba primero y, en lugar de fallar desde un primer momento, hace que la prueba pase desde el principio (excepto la primera prueba en la clase de prueba).

Cuando es así, se revela el *Success Against All Odds*. La práctica de empezar la prueba haciéndola pasar desde un inicio hace que el test pase incluso cuando se espera que falle.

---

51 https://www.codurance.com/es/publications/tdd-and-anti-patterns-chapter-5

Para representar este escenario, en el siguiente snippet se intenta implementar un repositorio desde SpringBoot que pagina y hace consultas basándose en un query string dado.

Nota: Para simplificar el ejemplo, se ha suprimido el teardown. El teardown elimina todos los datos insertados en la base de datos utilizada durante la prueba.

```
@Repository
class ProductsRepositoryWithPostgres(
    private val entityManager: EntityManager
) : Repository {

    override fun listFilteredProducts(query: String?, input: PagingQueryInput?)
{
        val pageRequest: PageRequest = input.asPageRequest()
        val page: Page<Product> = if (query.isNullOrBlank()) {
            entityManager.findAll(pageRequest)
        } else {
            entityManager.findAllByName(query, pageRequest)
        }
        return page
    }
}
```

Después de observar el código que da el acceso a la base de datos y de aplicar los criterios, testeamos el repositorio con el siguiente código de prueba.

Desde el principio, hemos estado haciendo algunas operaciones pesadas para poblar la base de datos con diferentes datos. Esto podría ser potencialmente un código que no huele muy bien.

```
private fun setupBeforeAll() {
    productIds = (1..100).map { db().productWithDependencies().apply().
    get<ProductId>() }
    productIdsContainingWood.addAll(
        (1..3).map { insertProductWithName("WoodyWoodOrange " + faker.
        funnyName().name()) }
    )
    productIdsContainingWood.addAll(
        (1..3).map {
            insertProductWithName(
                faker.funnyName().name() + " WoodyWoodOrange " + faker.
                funnyName().name()
            )
```

```
11          }
12      )
13  }
```

Una vez que la configuración esté hecha, veremos el primer caso de prueba de esta clase. El objetivo del caso de prueba es comprobar que dado el parámetro de ordenación CREATED_AT_ASC (línea 4), que es el que estamos buscando, los datos se ordenen en correspondencia con el.

```
1   @Test
2   fun `list products sorted by creation at date ascending`() {
3       val pageQueryInput = PagingQueryInput(
4           size = 30, page = 0, sort = listOf(Sort.CREATED_AT_ASC)
5       )
6       val result = repository.listFilteredProducts("", pageQueryInput)
7
8       assertThat(result.currentPage).isEqualTo(0)
9       assertThat(result.totalPages).isEqualTo(4)
10      assertThat(result.totalElements).isEqualTo(112)
11
12      assertThat(result.content.size).isEqualTo(30)
13      assertThat(result.content).allSatisfy { productIds.subList(0, 29).
        contains(it.id) }
14  }
```

Veamos un poco lo que está ocurriendo en el código guiándonos por las líneas:

1. Línea 3, 4 y 5: El parámetro que enviamos al repositorio con el orden que queremos y la paginación.
2. Línea 6: La ejecución del código que queremos probar.
3. Línea 8: Comprobamos que la página que volvió del repositorio sea la primera.
4. Línea 9: Verificamos que hay 4 páginas en total.
5. Línea 10: Comprobamos que hay 112 en total.
6. Línea 12: Comprobamos que la lista de elementos devueltos es la misma que se solicitó en la paginación.
7. Línea 13: Verificamos que la lista que da de vuelta coincide con aquella creada en la configuración anteriormente.

El siguiente caso de texto muestra una variante de lo que puede que queramos probar (el orden inverso). En lugar del orden ascendente, ahora probaremos el orden descendente. Si te fijas, la mayoría de las validaciones son las mismas si se comparan con las del caso de prueba anterior (desde la línea 6 a la 14).

```
1  @Test
2  fun `list products sorted by creation at date ascending`() {
3      val pageQueryInput = PagingQueryInput(
4          size = 30, page = 0, sort = listOf(Sort.CREATED_AT_DESC)
5      )
6      val result = repository.listFilteredProducts("", pageQueryInput)
7
8      assertThat(result.currentPage).isEqualTo(0)
9      assertThat(result.totalPages).isEqualTo(4)
10     assertThat(result.totalElements).isEqualTo(112)
11
12     assertThat(result.content.size).isEqualTo(30)
13     assertThat(result.content).allSatisfy { productIds.subList(0, 29).
contains(it.id) }
14 }
```

Para evitar repetir la lista anterior, nos concentraremos en los puntos importantes.

El primer aspecto relevante es el número de validaciones que podríamos no necesitar para cada caso de prueba. Por ejemplo, de la línea 8 a la 12, algunas validaciones verifican la paginación y los números relacionados con la lista leyendo el nombre del test. Nuestro objetivo es comprobar primero el orden y no la paginación. En pocas palabras, podríamos haber utilizado sólo la última validación para esta prueba.

Para continuar, veremos un poco más en detalle la línea 13. Tener una validación como la que se encuentra en esta línea es una de las posibles causas del *Success Against All Odds*. De hecho, en el código de prueba, es una de ellas, ya que afirma un subconjunto de la lista que siempre será verdadero.

En el libro de patrones xUnit, una forma de evitar este comportamiento falso/positivo es mantener el código lo más simple posible, sin lógica en él[52]. Esto se denomina prueba robusta (Meszaros 2007).

Refactorizar el Success Against All Odds

La pregunta aquí es, ¿qué podríamos hacer para evitar esto? Una posible solución para este caso de prueba y el código fuente se relaciona con la división de responsabilidades en el caso de prueba. Podríamos centrarnos sólo en el orden y luego probar la paginación con un test sucesivo.

El primer ejemplo sería poner la lista en orden ascendente. Vale la pena

---

52 El Free Ride descrito en la sección 8.4 también usa lógica dentro de los casos de prueba.

mencionar que con este enfoque, potencialmente podríamos eliminar la gran configuración mostrada anteriormente en el hook *setupBeforeAll*. Para esta aproximación, configuramos los datos requeridos para la prueba dentro de ella. De este modo, las pruebas no comparten estado entre sí.

```
@Test
fun `list products sorted by ascending creation date`() {
    db().productWithDependencies("created_at" to "2022-04-03T00:00:00.00Z").
    apply() // 1
    db().productWithDependencies("created_at" to "2022-04-02T00:00:00.00Z").
    apply() // 2
    db().productWithDependencies("created_at" to "2022-04-01T00:00:00.00Z").
    apply() // 3

    val pageQueryInput = PagingQueryInput(sort = listOf(SortOrder.CREATED_AT_
    ASC))

    val result = repository.listFilteredProducts("", pageQueryInput)

    assertThat(result.content[0].createdAt).isEqualTo("2022-04-01T00:00:00.00Z")
    assertThat(result.content[1].createdAt).isEqualTo("2022-04-02T00:00:00.00Z")
    assertThat(result.content[2].createdAt).isEqualTo("2022-04-03T00:00:00.00Z")
}
```

Una vez hecho esto, pasamos al caso de prueba de orden descendente, que es el mismo solo que la validación y la configuración han cambiado:

```
@Test

  fun `list products sorted by creation at date descending`() {
      db().productWithDependencies("created_at" to "2022-04-01T00:00:00.00Z").
      apply()
      db().productWithDependencies("created_at" to "2022-04-02T00:00:00.00Z").
      apply()
      db().productWithDependencies("created_at" to "2022-04-03T00:00:00.00Z").
      apply()

      val pageQueryInput = PagingQueryInput(sort = listOf(SortOrder.CREATED_AT_
      DESC))

      val result = repository.listFilteredProducts("", pageQueryInput)

      assertThat(result.content[0].createdAt).isEqualTo("2022-04-
```

```
        03T00:00:00.00Z")
12      assertThat(result.content[1].createdAt).isEqualTo("2022-04-
        02T00:00:00.00Z")
13      assertThat(result.content[2].createdAt).isEqualTo("2022-04-
        01T00:00:00.00Z")
14  }
```

Lo siguiente es la paginación. Ahora podemos empezar a centrarnos en la paginación y los aspectos que aporta.

Una vez que tenemos el orden listo, podemos empezar a ver la paginación y, por supuesto, tratar de probar una cosa específica a la vez. El siguiente ejemplo muestra cómo podríamos confirmar que tenemos el número deseado de páginas al paginar el resultado.

```
1   @Test
2     fun `should have one page when the list is ten`() {
3         insertTenProducts()
4         val page = PagingQueryInput(size = 10)
5
6         val result = repository.listFilteredProducts(
7             null,
8             null,
9             Page
10        )
11
12        assertThat(result.totalPages).isEqualTo(1)
13  }
```

El enfoque de descomponer las pruebas en "unidades"[53] más pequeñas ayudaría a la comunicación entre el equipo que se ocuparía de este código más adelante y también haría que las pruebas fuesen más robustas.

Aspectos a tener en cuenta

1. Empieza con la prueba en rojo siempre que sea posible.
2. Evita repetir las mismas afirmaciones de casos de prueba anteriores.
3. Evita que los casos de prueba compartan estado entre sí.

---

53 Aquí la unidad no se refiere a una función o método sino a un comportamiento (o responsabilidad).

El Stranger

Es un caso de prueba que ni siquiera pertenece a la prueba unitaria de la que forma parte. En realidad, se prueba un objeto separado, probablemente un objeto que es utilizado por el objeto bajo prueba. Pero aquí, el caso de prueba testea ese objeto directamente sin depender del output del objeto bajo prueba, haciendo uso de ese objeto para su propio comportamiento. También es conocido como TheDistantRelative (Carr 2022).

Discutimos el *Stranger* en el Episodio 5 de la serie de vídeos[54] sobre antipatrones de TDD organizada por Codurance.

Empecemos con una introducción. En este blog de java revisited[55], la forma en que se explica la ley de Deméter nos da una pista de por qué el *Stranger* es un antipatrón. También podemos relacionarlo con el libro Clean Code, que recomienda "hablar con amigos, no con extraños" a la hora de diseñar código. En el ejemplo dado en el libro, la cadena de métodos es la que más expone al *Stranger*. El ejemplo se utiliza en código de producción.

Carlos Caballero, en su entrada de blog "La ley de Deméter: No hables con extraños"[56], también utiliza código de producción para dar un ejemplo de violación de la ley. Él proporciona un code snippet que idealmente debería probarse. Es en este punto donde nos extenderemos y, en concreto, implementaremos el código de prueba de apoyo.

Para empezar, aquí está el código que representa la violación de la ley de Deméter dentro del código de producción:

```
person
    .getHouse() // return an House's object
    .getAddress() // return an Address's object
    .getZipCode() // return a ZipCode Object
```

Este tipo de código podría potencialmente conllevar al antipatrón el *Stranger* dentro del código de prueba. Por ejemplo, para comprobar que una persona tiene un código postal válido, podríamos escribir algo como esto:

```
describe('Person', () => {
    it('should have valid zip code', () => {
        const person = FakeObject.createAPerson({ zipCode: '56565656' });
        person
```

54 https://www.codurance.com/es/publications/tdd-and-anti-patterns-chapter-5

55 https://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html

56 https://betterprogramming.pub/demeters-law-don-t-talk-to-strangers-87bb4af11694

```
            .getHouse()
            .getAddress()
            .getZipCode()
        expect('56565656').toEqual(person.house.address.zipCode);
    });
});
```

Si queremos acceder al código postal, tenemos que ir hasta el objeto ZipCode. Esto nos da una pista de que en realidad lo que queremos comprobar es el propio objeto Address y no Person.[57]

```
describe('Address', () => {
    it('should have valid zip code', () => {
        const address = new Address(
            '56565656',
            '1456',
            'Street X',
            'My city',
            'Great state',
            'The best country'
        );
        expect('56565656').toEqual(address.getZipCode());
    });
});
```

La prueba en sí tiene algo que podría mejorarse para evitar esto. Por ejemplo, la interacción entre el objeto Person, Address y Zip code podría "ocultarse" dentro de una implementación detrás de una abstracción. En este caso, en lugar de navegar directamente por todo el gráfico de objetos, probaríamos el output de esto.

Antes de pasar al siguiente antipatrón, recuerda que el *Stranger* también podría clasificarse como un test que apesta. He aquí algunas señales que podrían conducir al *Stranger*:

1. Se relaciona con el patrón xUnit (Meszaros 2007) en la sección "Test smells".
2. El uso de mocking.

57 La idea no es que el tema gire en torno a la relación 1-1 entre el código de producción y el código de prueba, sino más bien que represente un escenario específico en el que también estamos rompiendo la encapsulación.

La Hidden Dependency

Es un primo hermano del *Local Hero*. La *Hidden Dependency* es una prueba unitaria que requiere que algunos datos existentes hayan sido rellenados en algún lugar antes de que la prueba se ejecute. Si esos datos no se rellenan, la prueba fallaría y le dejaría pocas indicaciones a la persona que está desarrollando software sobre lo que quiere o por qué, y obligándoles a revisar hectáreas de código para averiguar de dónde se supone que vienen los datos que está utilizando (Carr 2022).

Discutimos la *Hidden Dependency* en el Episodio 4 de la serie de vídeos[58] sobre antipatrones de TDD organizado por Codurance.

La *Hidden Dependency* es un antipatrón muy popular entre los profesionales. En concreto, la *Hidden Dependency* estorba y hace que los profesionales estén insatisfechos con el testing en términos generales. Esta puede ser la causa de horas de depuración de código de prueba para entender por qué falla un test. A veces da poca o ninguna información sobre la causa principal. Este problema está relacionado con lo siguiente:

- Bases de datos (integradas para realizar pruebas)
- Builders (lógica compleja para construir datos para configurar un caso de prueba)

En la siguiente sección, vamos a repasar la librería de gestión de estados Vuex que esconde la complejidad de manejar datos para aplicaciones front-end. Si no estás familiarizado con Vuex[59] o el patrón Flux, es recomendable que lo revises primero.

La Vuex Dependency

El ejemplo de esta sección está relacionado con Vue y Vuex. En este caso de prueba, el objetivo es listar usuarios en una lista desplegable. Vuex se utiliza como una 'fuente de la verdad' para los datos.

```
1  export const Store = () => ({
2    modules: {
3      user: {
4        namespaced: true,
5        state: {
6          currentAdmin: {
7            email: 'fake@fake.com',
```

58 https://www.codurance.com/es/publications/tdd-and-anti-patterns-chapter-4

59 https://vuex.vuejs.org

```
8           },
9         },
10        getters: userGetters,
11      },
12      admin: adminStore().modules.admin,
13    },
14  });
```

En la línea 2, se define la estructura necesaria para Vuex, y en la línea 12 se crea el admin store. Una vez que la stubbed store está lista, podemos empezar a escribir la prueba en sí. Como una pista para la siguiente pieza de código, ten en cuenta que la tienda no tiene parámetros.

```
1   it('should list admins in the administrator field to be able to pick one up',
    async () => {
2     const store = Store();
3
4     const { findByTestId, getByText } = render(AdminPage as any, {
5       store,
6       mocks: {
7         $route: {
8           query: {},
9         },
10      },
11    });
12    await fireEvent.click(await findByTestId('admin-list'));
13    await waitFor(() => {
14      expect(getByText('Admin')).toBeInTheDocument();
15    });
16  });
```

En la línea 2 creamos el store para usarlo en el código bajo prueba, y en la línea 14, intentamos buscar el texto Admin. Asumimos que la lista funciona si está en el texto.

Aquí el truco está en que si la prueba no encuentra el Admin, tendremos que revisar el código dentro del store para ver lo que está pasando. Como te habrás dado cuenta, si nos fijamos sólo en el caso de prueba, no está claro de dónde viene el texto Admin.

El siguiente ejemplo de código muestra un mejor enfoque para utilizar explícitamente los datos necesarios al configurar la prueba. Esta vez, en la línea 2 se espera que el Admin exista de antemano.

```
it('should list admins in the administrator field to be able to pick one up',
async () => {
  const store = Store({ admin: { name: 'Admin' } });

  const { findByTestId, getByText } = render(AdminPage as any, {
    store,
    mocks: {
      $route: {
        query: {},
      },
    },
  });

  await fireEvent.click(await findByTestId('admin-list'));

  await waitFor(() => {
    expect(getByText('Admin')).toBeInTheDocument();
  });
});
```

En general, la *Hidden Dependency* puede aparecer de diferentes maneras y en diferentes estilos de pruebas. En la siguiente sección se describe un problema oculto que surge al probar la integración con una base de datos.

La dependencia de base de datos

Aquí estamos intentando obtener compras manuales de la base de datos basándonos en un criterio dado que se implementa detrás del método *get_manual_purchases*. A continuación, comparamos lo obtenido por el método con el resultado deseado, que se almacena en un archivo CSV.

```

def test_dbdatasource_is_able_to_load_products_related_only_to_manual_purchase(
self, db_resource
):
config_file_path = Path("./tests/data/configs/docker_config.json")
expected_result = pd.read_csv("./tests/data/manual_product_info.csv")
datasource = DBDataSource(config_file_path=config_file_path)
result = datasource.get_manual_purchases()
assert result.equals(expected_result)

```

En las líneas 4 y 5 el *setup* se realiza a través de archivos de configuración, la

línea 5 es importante ya que el resultado de la prueba debe coincidir con su contenido. A continuación, en la línea 9, se ejecuta el código bajo prueba.

Dentro de este método, hay una consulta que se ejecuta con el fin de obtener las compras manuales y afirmar que es igual al resultado esperado:

```
1  query: str = """
2  select
3  product.id
4  po.order_id,
5  po.quantity,
6  product.country
7  from product
8  join purchased as pur on pur.product_id = product.id
9  join purchased_order as po on po.purchase_id = cur.id
10 where product.completed is true and
11 pur.type = 'MANUAL' and
12 product.is_test is true
13 ;
14 """
```

Esta consulta tiene una cláusula *where* particular que se oculta del caso de prueba, lo que provoca que la prueba falle. Por defecto, los datos generados a partir del resultado esperado establecen la *flag* test como falsa, lo cual no devuelve ningún resultado en el caso de prueba.

Aspectos a tener en cuenta

1. Probar la integración de datos lo antes posible.
2. Si es posible, evitar el uso de datos de fuentes externas en las pruebas.

El Greedy Catcher

Una prueba unitaria que captura excepciones y elimina el seguimiento del stack, a veces sustituyéndolo por un mensaje de error menos informativo, pero en ocasiones incluso simplemente registrando y dejando pasar la prueba, por ejemplo, *Loudmouth*, (Carr 2022).

El *Greedy Catcher* forma parte del Episodio 4 de la serie de videos[60] sobre los antipatrones de TDD, presentada por Codurance.

---

60 https://www.codurance.com/es/publications/tdd-and-anti-patterns-chapter-4

Manejar excepciones (o incluso utilizarlas) puede ser complicado. Algunos profesionales abogan por no usar excepciones en absoluto[61]; otros las utilizan como mecanismo para informar de que algo ha ido mal durante la ejecución del programa.

A pesar del tipo de profesional de desarrollo de software que seas, probar excepciones puede desvelar algunos patrones que perjudican al enfoque de *test-first*.[62]

El Greedy Catcher aparece cuando el elemento bajo prueba maneja la excepción y oculta información útil sobre el tipo de excepción, el mensaje o la trayectoria de la pila. Dicha información es necesaria para mitigar el lanzamiento de posibles excepciones no deseadas.

A continuación, tenemos un ejemplo de un posible candidato para El *Greedy Catcher*. Este trozo de código fue extraído del proyecto *Laravel/Cashier stripe* – *Laravel* está escrito en PHP y es uno de los proyectos más populares dentro de este ecosistema.

El siguiente código es un paquete que envuelve el SDK[63] de Stripe en un paquete *Laravel* que ofrece un enfoque más fácil para integrar *stripe* en aplicaciones *Laravel*.

El proyecto Laravel/Cashier Stripe

A pesar de tener un manejador *try/catch* dentro del caso de prueba (que potencialmente podría apuntar a mejoras adicionales) cuando la excepción es lanzada, el caso de prueba la atrapa y activa cierta lógica:

```
1  public function test_retrieve_the_latest_payment_for_a_subscription()
2  {
3  $user = $this->createCustomer('retrieve_the_latest_payment_for_a_subscription');
4  try {
5  $user->newSubscription('main', static::$priceId)
6  ->create('pm_card_threeDSecure2Required');
7  $this->fail('Expected exception '.IncompletePayment::class.' was not thrown.');
8  } catch (IncompletePayment $e) {
```

61 Aquí hay un hilo de StackOverflow que trata el tema en profundidad https://stackoverflow.com/questions/1736146/why-is-exception-handling-bad.

62 The Greedy Catcher y The secret Catcher son similares pero cubren diferentes aspectos de las excepciones. En la sección 9.6 cubrimos en detalle The Secret Catcher.

63 Stripe Software Development Kit – https://stripe.com/docs/development/quickstart/php.

```
9   $subscription = $user
10  ->refresh()
11  ->subscription('main');
12  $this->assertInstanceOf(
13  Payment::class,
14  $payment = $subscription->latestPayment()
15  );
16  $this->assertTrue($payment->requiresAction());
17  }
18  }
19
```

El *Greedy Catcher* surge no sólo en el código de prueba, sino también en el código de producción, ocultando información útil para rastrear una excepción y supone una fuente de pérdida de tiempo que podría ahorrarse si el código se escribe de otra manera.

En la siguiente sección, veremos un ejemplo de cómo el antipatrón *Greedy Catcher* también puede encontrarse dentro del código de producción.

Análisis del token JTW con JavaScript

El siguiente ejemplo es una representación de un componente *middleware* JavaScript que analiza un token JWT y redirige al usuario si el token está vacío. El código utiliza las librerías *Jest y Nuxtjs*.

La línea 3 decodifica el token JWT a través del paquete *jwt-decode*, en caso de éxito, el middleware sigue el flujo. il y si el token es falso por cualquier razón, invoca la función de cierre de sesión (en las líneas 8 y 11).

Por el aspecto del código, es difícil reconocer que bajo el bloque catch, se está ignorando la excepción y, si ocurre algo, el resultado será lo que devuelva la función logout (línea 11).

```
1   export default function(context: Context) {
2   try {
3   const token = jwt_decode(req?.cookies['token']);
4   if (token) {
5   return null;
6   } else {
7   return await logout($auth, redirect);
8   }
9   } catch (e) {
```

```
10 return await logout($auth, redirect);
11 }
12 }
13
```

El código de prueba utiliza algún contexto *Nuxtjs* para crear la solicitud que va a ser procesada por el *middleware*. El caso único de prueba muestra un enfoque para verificar si el usuario está siendo deslogueado o si el token no es válido. Observa que la cookie está detrás de la variable *serverParameters*.

```
1  it('should log out when token is invalid', async () => {
2  const redirect = jest.fn();
3  const serverParameters: Partial<IContextCookie> = {
4  route: currentRoute as Route, $auth, redirect, req: { cookies: null },
5  };
6  await actions.nuxtServerInit(
7  actionContext as ActionContext,
8  serverParameters as IContextCookie
9  );
10 expect($auth.logout).toHaveBeenCalled();
11 });
12
```

La trampa está en que el test de arriba funciona como debería, pero no por la razón esperada. *serverParameters* contiene el objeto *req* que tiene las cookies establecidas en *null* (línea 3). Cuando ese es el caso, JavaScript lanzará un error ya que no será posible acceder a un token de *null*.[64]

Este comportamiento ejecuta el bloque catch, que llama a la función de cierre de sesión (línea 12). El *stack trace* de este error no aparecerá en ningún sitio, ya que el bloque catch ignora la excepción en el código de producción.

Aspectos a tener en cuenta

1. Ocultar información en bloques *try/catch* dificulta la detección de problemas en el código de producción.
2. Es importante dedicar tiempo a entender por qué la prueba no está generando el comportamiento deseado.

El Peeping Tom

Un *test* que debido a recursos compartidos puede ver los datos de resultado

64 Javascript se comportará así ante un intento de acceder a cualquier propiedad desde una variable que sea nula o indefinida.

de otro test, y puede hacer que falle aunque el sistema bajo prueba sea perfectamente válido. Esto se ha visto comúnmente en *fitness*, donde el uso de variables miembro estáticas para mantener colecciones no se limpian adecuadamente después de la ejecución de la prueba y aparecen inesperadamente en otras ejecuciones de prueba. También conocido como *The Uninvited Guests* (Carr 2022).

El *Peeping Tom* forma parte del Episodio 6 de la serie videos[65] sobre los antipatrones de TDD, presentada por Codurance.

Tener que tratar con cualquier estado global dentro de un caso de prueba es algo que aporta una capa extra de complejidad. Requiere pasos de limpieza explícitos antes de cada prueba e incluso después de su ejecución para evitar efectos secundarios.

El *Peeping Tom* representa el problema al que nos enfrentamos cuando utilizamos cualquier forma de estado global durante la ejecución de pruebas. En *Stackoverflow*, el popular sitio web de preguntas y respuestas, hay un hilo dedicado a este tema con algunos comentarios que ayudan a entenderlo mejor. Christian Posta también escribió en su blog que los métodos estáticos son *smells* de código.[66]

Allí, hay un fragmento que se extrajo de esta publicación que describe cómo el uso de *Singleton* (Gamma et al. 1994) y las propiedades estáticas pueden dañar el caso de prueba y mantener el estado entre las pruebas. Aquí, vamos a utilizar el mismo ejemplo con pequeños cambios para hacer que el código compile.

La idea detrás del *Singleton* es crear y reutilizar una única instancia a partir de cualquier tipo de objeto.[67] Así que para conseguirlo, podemos crear una clase (en este ejemplo llamada *MySingleton*) y bloquear la creación de un objeto a través de su constructor y permitir sólo la creación dentro de la clase, controlada por el método *getInstance*:

```
1  public class MySingleton {
2  private static MySingleton instance;
3  private String property;
4  private MySingleton(String property) {
5  this.property = property;
```

65 https://www.codurance.com/es/publications/tdd-and-anti-patterns-capitulo-6

66 La entrada del blog se puede encontrar en https://blog.christianposta.com/testing/java-static-methods-can-be-a-code-smell.

67 Rainer Grimm enumeró las ventajas y desventajas del Singleton, allí, enumeró el acceso global como una ventaja, por lo tanto, también mencionó la cuestión relativa a la testeabilidad que relaciona con La Dependencia Oculta que discutimos en el Capítulo 9.3.

```
6  }
7  public static synchronized MySingleton getInstance() {
8  if (instance == null) {
9  instance = new MySingleton(System.getProperty("com.example"));
10 }
11 return instance;
12 }
13 public Object getSomething() {
14 return this.property;
15 }
16 }
```

Cuando se trata de testear, no hay mucho en el sentido de una interfaz pública con la que podamos interactuar. Sin embargo, el método expuesto en el *MySingleton* llamado *getSomething* puede ser invocado y afirmado contra un valor como se muestra en el siguiente fragmento:

```
1  import org.junit.jupiter.api.Test;
2  import static org.assertj.core.api.Assertions.assertThat;
3  class MySingletonTest {
4  @Test
5  public void somethingIsDoneWithAbcIsSetAsASystemProperty(){
6  System.setProperty("com.example", "abc");
7  MySingleton singleton = MySingleton.getInstance();
8  assertThat(singleton.getSomething()).isEqualTo("abc");
9  }
10 }
```

Un caso de prueba simple pasará sin ningún problema, el caso de prueba crea la instancia *Singleton* e invoca el *getSomething* para recuperar el valor de la propiedad definida cuando se definió la prueba. El problema surge cuando intentamos probar el mismo comportamiento pero con diferentes valores en el *System.setProperty*.

```
1  import org.junit.jupiter.api.Test;
2  import static org.assertj.core.api.Assertions.assertThat;
3  class MySingletonTest {
4  @Test
5  public void somethingIsDoneWithAbcIsSetAsASystemProperty(){
6  System.setProperty("com.example", "abc");
7  MySingleton singleton = MySingleton.getInstance();
8  assertThat(singleton.getSomething()).isEqualTo("abc");
9  }
```

```
@Test
public void somethingElseIsDoneWithXyzIsSetAsASystemProperty(){
System.setProperty("com.example", "xyz");
MySingleton singleton = MySingleton.getInstance();
assertThat(singleton.getSomething()).isEqualTo("xyz");
}
}
```

Dada la naturaleza del código, el segundo caso de prueba fallará y mostrará que sigue manteniendo el valor abc.

Como el *Singleton* garantiza una sola instancia de un objeto dado, durante la ejecución de la prueba, la primera prueba que se ejecuta crea la instancia y, para las siguientes ejecuciones, reutiliza la misma instancia creada anteriormente.

El problema aquí es "fácil" de ver ya que un caso de prueba se ejecuta después del otro. Pero, para los marcos de pruebas que ejecutan tests en paralelo o que no garantizan su orden (que es a menudo el comportamiento por defecto), puede ser mucho más difícil identificar los fallos de las pruebas causados por una dependencia de un estado global que ha sido mutado por otras pruebas.

Puede confundir a los profesionales porque la prueba pasará cuando se ejecute sola y fallará cuando se ejecute junto con otras.

Como la clase *Singleton* tiene una propiedad privada que controla la instancia creada, no es posible limpiarla sin cambiar el propio código (lo que sería un cambio sólo a efectos de pruebas). Por lo tanto, otro enfoque sería utilizar la reflexión para restablecer la propiedad y empezar siempre con una instancia nueva, como muestra el siguiente código:

```
class MySingletonTest {
@BeforeEach
public void resetSingleton() throws SecurityException, NoSuchFieldException,
IllegalArgumentException, IllegalAccessException {
Field instance = MySingleton.class.getDeclaredField("instance");
instance.setAccessible(true);
instance.set(null, null);
}
}
```

Haciendo uso de *reflection*, es posible restablecer la instancia antes de que se ejecute cada caso de prueba (utilizando la anotación *@BeforeEach)*. Aunque este enfoque es posible, debe llamar la atención el código extra y los posibles efectos secundarios al probar la aplicación utilizando dicho patrón.

Representar así a El *Peeping Tom*, además de tener que usar reflection para resetear una propiedad, puede no parecer perjudicial para testear, pero puede complicarse cuando una pieza de código que queremos testear depende de un *Singleton*.

Como bien comparte Rodaney Glitzel (2022), el problema no es *Singleton* en sí, sino el código que depende de ello, ya que se vuelve más difícil de testear.

El Secret Catcher

Un test que a primera vista parece no estar haciendo ninguna prueba debido a la ausencia de aserciones, pero ya sabemos que, "el diablo está en los detalles". La prueba depende de que se lance una excepción cuando ocurra un imprevisto, y espera que el marco de pruebas capture la excepción y la comunique al usuario como un fallo (Carr 2022).

El *Secret Catcher* forma parte del Episodio 3 de la serie videos[68] sobre los antipatrones de TDD, presentada por Codurance.

El *Secret Catcher* es un viejo amigo de los desarrolladores. Se puede detectar en diferentes bases de código. Este antipatrón a menudo se relaciona con la "prisa" en la que las funcionalidades deben ser entregadas o incluso con el deseo de alcanzar un X% de cobertura de pruebas.

Este antipatrón esconde un secreto en su nombre porque el código no deja claro que se supone que debe lanzar una excepción. En lugar de manejar (o capturar) dicha excepción, el caso de prueba simplemente la ignora. El *Secret Catcher* también está relacionado con El *Greedy Catcher*.

El siguiente ejemplo escrito en Vue.js con un cliente Apollo es un intento de representar tal escenario. A primera vista, el método parece correcto y hace lo que se espera de él. En otras palabras, envía una operación de mutación al servidor para eliminar el tipo de pago del usuario asociado y, al final, actualiza la interfaz de usuario:

```
1  async removePaymentMethod(paymentMethodId: string) {
2  this.isSavingCard = true;
3  const { data } = await this.$apolloProvider.clients.defaultClient.mutate({
4  mutation: DetachPaymentMethodMutation,
5  variables: { input: { stripePaymentMethodId: paymentMethodId } },
6  });
```

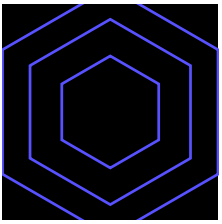
68 https://www.codurance.com/es/publications/tdd-y-anti-patrones-capitulo-3

```
7  if (this.selectedCreditCard === paymentMethodId) {
8  this.selectedCreditCard = null;
9  }
10 this.isSavingCard = false;
11 }
```

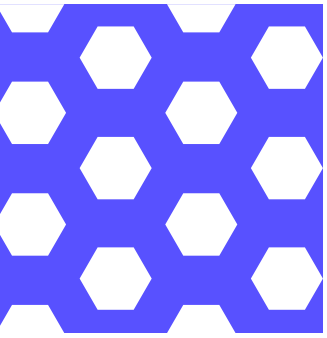
La prueba *JavaScript* está escrita usando *jest* y la librería de *testing*. Pero, antes de seguir adelante con el texto, tenemos un reto para ti, ¿puedes detectar lo que falta en el caso de prueba?

```
1  test('it handles error when removing credit card ', async () => {
2  const data = await Payment.asyncData(asyncDataContext);
3  data.paymentMethod = PaymentMethod.CREDIT_CARD;
4  const { getAllByText } = render(Payment, {
5  mocks,
6  data() {
7  return { ...data };
8  },
9  });
10 const [removeButton] = getAllByText(Remove');
11 await fireEvent.click(removeButton);
12 });
13
```

Empecemos por la aserción que falta al final del caso de prueba. Si no te has dado cuenta, el último paso que hace este *test* es esperar al evento *click*. Para alguna funcionalidad que elimine un método de pago de un usuario, aseverar que se muestra un mensaje sería una buena idea.



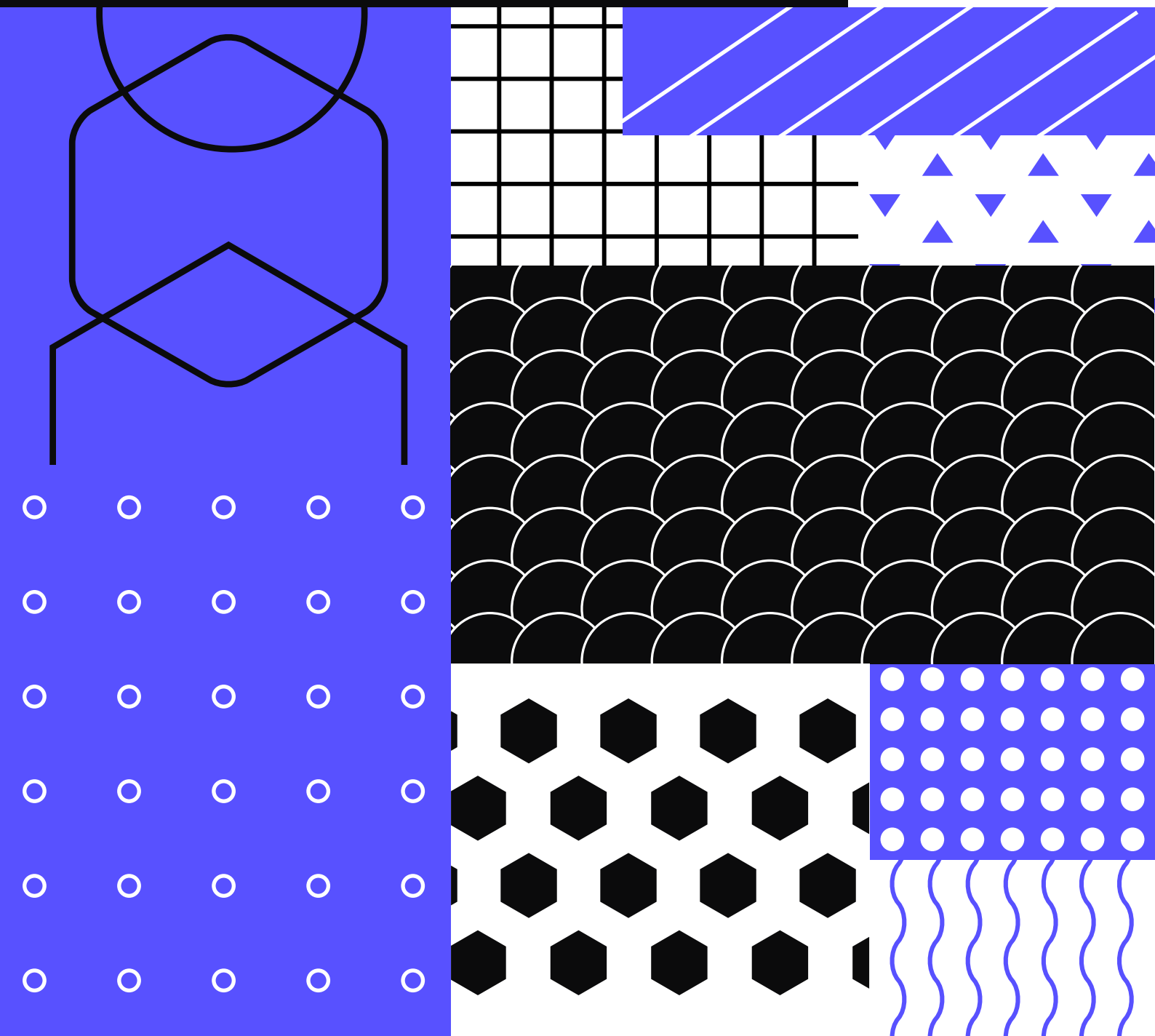
Además, el caso de prueba se basa en una configuración específica que asume que la mutación *Graphql* funcionará siempre. Sin embargo, es perfectamente posible que la mutación *Graphql* falle y lance una excepción dentro del código de producción. Pero la prueba está diseñada para hacer frente a esa situación. En este escenario, el caso de prueba se basa en el marco de pruebas *jest* para informar del error, si lo hay, en lugar de que el error sea manejado dentro del propio test.



Aspectos a tener en cuenta

1. Evitar confiar en los errores del test runner.
2. Hacer explícita la aserción deseada en el caso de prueba.

Nivel III



Nivel III

En este capítulo, revisaremos las prácticas a las que nos podemos enfrentar cuando llevamos un tiempo aplicando TDD y, en algunos casos, incluso cuando intentamos utilizar TDD en bases de código que no están preparadas para la testabilidad.

- Evitar tener un caso de prueba que haga todo simultáneamente, lo cual produce muchas líneas en un solo caso de prueba.
- Pasar demasiado tiempo configurando el caso de prueba indica que el código no está diseñado para la testabilidad, esto se relaciona con El *Mockery*, tratado más adelante.
- Si es posible, evitar infringir la encapsulación para alcanzar el 100% de la cobertura del código.

El Giant

Un *test* unitario que, aunque está probando correctamente el objeto sometido a prueba, puede abarcar miles de líneas y contener muchísimos casos de prueba. Esto puede indicar que el sistema sometido a prueba es un *God Object* (Carr 2022).

El *Giant* forma parte del Episodio 1 de la serie videos[69] sobre los antipatrones de TDD, presentada por Codurance.

El antipatrón *Giant* también es una señal de que algo falta en el diseño del *codebase*. El diseño del código es a menudo un tema de discusión entre los practicantes de TDD (Mancuso 2018). Similar al *Excessive Setup*, este antipatrón también puede ocurrir mientras se desarrolla con TDD. El *Giant* suele estar relacionado con el diseño de código de la clase *God*. Este es un antipatrón para la Programación Orientada a Objetos y también va en contra de los principios SOLID (*Single responsibility, Open-Close, Liskov substitution, Interface segregation and Dependency Inversion*) (Martin 2017; (Stemmler 2022).

El proyecto Nuxtjs

En TDD, El Giant a menudo se muestra con muchas aserciones en un solo caso de prueba. Dave Farley lo demuestra en su vídeo. El mismo archivo de prueba utilizado de Nuxtjs[70] [71]*El Excessive Setup* muestra signos de El *Giant*. Si

69 https://www.codurance.com/es/publications/tdd-anti-patterns-chapter-1

70 https://nuxtjs.org

71 https://github.com/nuxt/nuxt.js/blob/d4b9e4b0553bcd617ecbc0b8b76871070b347fcb/packages/server/test/server.test.js#L166

inspeccionamos el código más de cerca, vemos un trozo de código seguido de aserciones, luego más código también seguido de más aserciones y todo dentro del mismo caso de prueba:

```
1  test('should setup middleware', async () => {
2  const nuxt = createNuxt()
3  const server = new Server(nuxt)
4  server.useMiddleware = jest.fn()
5  server.serverContext = { id: 'test-server-context' }
6  await server.setupMiddleware()
7  expect(server.nuxt.callHook).toBeCalledTimes(2)
8  expect(server.nuxt.callHook).nthCalledWith(1, 'render:setupMiddleware', server.
   app)
9  expect(server.nuxt.callHook).nthCalledWith(2, 'render:errorMiddleware', server.
   app)
10 expect(server.useMiddleware).toBeCalledTimes(4)
11 expect(serveStatic).toBeCalledTimes(2)
12 expect(serveStatic).nthCalledWith(1, 'resolve(/var/nuxt/src, var/nuxt/static)',
   server.options.render.static)
13 expect(server.useMiddleware).nthCalledWith(1, {
14 dir: 'resolve(/var/nuxt/src, var/nuxt/static)',
15 id: 'test-serve-static',
16 prefix: 'test-render-static-prefix'
17 })
18 expect(serveStatic).nthCalledWith(2, 'resolve(/var/nuxt/build, dist, client)',
   server.options.render.dist)
19 expect(server.useMiddleware).nthCalledWith(2, {
20 handler: {
21 dir: 'resolve(/var/nuxt/build, dist, client)',
22 id: 'test-serve-static'
23 },
24 path: '__nuxt_test'
25 })
26 const nuxtMiddlewareOpts = {
27 options: server.options,
28 nuxt: server.nuxt,
29 renderRoute: expect.any(Function),
30 resources: server.resources
31 }
32 expect(nuxtMiddleware).toBeCalledTimes(1)
33 expect(nuxtMiddleware).toBeCalledWith(nuxtMiddlewareOpts)
34 expect(server.useMiddleware).nthCalledWith(3, {
```

```
35 id: 'test-nuxt-middleware',
36 ...nuxtMiddlewareOpts
37 })
38 const errorMiddlewareOpts = {
39 resources: server.resources,
40 options: server.options
41 }
42 expect(errorMiddleware).toBeCalledTimes(1)
43 expect(errorMiddleware).toBeCalledWith(errorMiddlewareOpts)
44 expect(server.useMiddleware).nthCalledWith(4, {
45 id: 'test-error-middleware',
46 ...errorMiddlewareOpts
47 })
48 })
```

El punto de atención aquí es reflexionar sobre si tiene sentido dividir cada bloque de código y aserción en su propio caso de prueba individual. La cuestión es tener un caso de prueba estructurado de forma que contenga un bloque de aserciones de código de prueba, seguido de más código de prueba y más aserciones posteriores.

Sería necesaria una revisión más detallada para comprobar si es posible romper el código como se ha sugerido anteriormente. Sin embargo, este escenario es un buen ejemplo de cómo puede manifestarse el antipatrón *Giant*. Como dice Dave Farley en su vídeo, esta práctica no es recomendable.

Brian Okken en *Python Testing with pytest* (Okken 2022) también esboza que tener una configuración de prueba con una estructura *Arrange-Assert-Act-Assert-Act-Assert* es un antipatrón. Argumenta que este estilo funciona hasta que el test falla. Esto se debe a que cuando falla, cualquiera de las acciones anteriores podría haber causado el error, lo que dificulta la detección rápida de la causa.

Aspectos a tener en cuenta

- Probar después, en lugar de probar primero.

El Excessive Setup

Una prueba que requiere mucho trabajo de configuración incluso para empezar a testear. A veces se utilizan varios cientos de líneas de código para configurar el entorno de una prueba, con diversos objetos implicados, lo que dificulta determinar realmente de lo que se está probando debido al "ruido" de todas las configuraciones en curso (Carr 2022).

El *Excessive Setup* forma parte del Episodio 1 de la serie videos[72] sobre los antipatrones de TDD, presentada por Codurance.

Las personas que desarrollan software pueden asociar este antipatrón a la ausencia de práctica de TDD desde el principio y a la falta de práctica de la calistenia de objetos.[73]

El planteamiento clásico para El *Excessive Setup* es cuando quieres probar un comportamiento específico dentro de tu código. Aún así, se vuelve difícil debido a las muchas dependencias que tienes que configurar primero. Cuando la cantidad de estas dependencias empiezan a dañar la testabilidad, es un *smell* de código.

El proyecto Nuxtjs

El siguiente código demuestra un caso de prueba del *framework Nuxtjs* que presenta este antipatrón. El archivo de prueba para el servidor comienza con algunos *mocks*, y luego continúa hasta el método *beforeEach* que contiene más trabajo de configuración (configuración de los *mocks*).

```
1  jest.mock('compression')
2  jest.mock('connect')
3  jest.mock('serve-static')
4  jest.mock('serve-placeholder')
5  jest.mock('launch-editor-middleware')
6  jest.mock('@nuxt/utils')
7  jest.mock('@nuxt/vue-renderer')
8  jest.mock('../src/listener')
9  jest.mock('../src/context')
10 jest.mock('../src/jsdom')
11 jest.mock('../src/middleware/nuxt')
12 jest.mock('../src/middleware/error')
13 jest.mock('../src/middleware/timing')
14 describe('server: server', () => {
15 const createNuxt = () => ({
16 options: {
17 dir: {
18 static: 'var/nuxt/static'
19 },
20 srcDir: '/var/nuxt/src',
21 buildDir: '/var/nuxt/build',
```

72 https://www.codurance.com/es/publications/tdd-anti-patterns-chapter-1

73 Object Calisthenics was introduced in the book The ThoughtWorks Anthology: Essays on Software Technology and Innovation(Steinberg 2008), in there nine steps to better software design were introduced.

```
22 globalName: 'test-global-name',
23 globals: { id: 'test-globals' },
24 build: {
25 publicPath: '__nuxt_test'
26 },
27 router: {
28 base: '/foo/'
29 },
30 render: {
31 id: 'test-render',
32 dist: {
33 id: 'test-render-dist'
34 },
35 static: {
36 id: 'test-render-static',
37 prefix: 'test-render-static-prefix'
38 }
39 },
40 server: {},
41 serverMiddleware: []
42 },
43 hook: jest.fn(),
44 ready: jest.fn(),
45 callHook: jest.fn(),
46 resolver: {
47 requireModule: jest.fn(),
48 resolvePath: jest.fn().mockImplementation(p => p)
49 }
50 })
51 beforeAll(() => {
52 jest.spyOn(path, 'join').mockImplementation((...args) => `join(${args.join(',
   ')})`)
53 jest.spyOn(path, 'resolve').mockImplementation((...args) => `resolve(${args.
   join(', ')})`)
54 connect.mockReturnValue({ use: jest.fn() })
55 serveStatic.mockImplementation(dir => ({ id: 'test-serve-static', dir }))
56 nuxtMiddleware.mockImplementation(options => ({
57 id: 'test-nuxt-middleware',
58 ...options
59 }))
60 errorMiddleware.mockImplementation(options => ({
61 id: 'test-error-middleware',
```

```
62  ...options
63  }))
64  createTimingMiddleware.mockImplementation(options => ({
65  id: 'test-timing-middleware',
66  ...options
67  }))
68  launchMiddleware.mockImplementation(options => ({
69  id: 'test-open-in-editor-middleware',
70  ...options
71  }))
72  servePlaceholder.mockImplementation(options => ({
73  key: 'test-serve-placeholder',
74  ...options
75  }))
76  })
77  })
```

Al leer el test desde el principio nos damos cuenta de que, para empezar, hay 13 invocaciones *jest.mock*. Además, hay más configuración en el *beforeEach*; alrededor de 9 *spies* y *stub setups*. Probablemente, si quisiéramos crear un nuevo caso de prueba desde cero o mover pruebas a través de diferentes archivos, necesitaríamos mantener la misma configuración excesiva tal y como está.

El proyecto testeable

El *Excessive Setup* es una trampa común. El siguiente código es de un proyecto de investigación llamado *Testable* (Marabesi y Silveira 2020) y muestra una única función que también sufre de muchas dependencias y lleva a una configuración excesiva para conseguir que la función se ejecute:

```
1  import { Component } from 'react';
2  import PropTypes from 'prop-types';
3  import { Redirect } from 'react-router';
4  import Emitter, { PROGRESS_UP, LEVEL_UP } from '../../../../packages/emitter/
   Emitter';
5  import { track } from '../../../../packages/emitter/Tracking';
6  import { auth } from '../../../../pages/login/Auth';
7  import Reason from '../../../../packages/engine/Reason';
8  import EditorManager from '../editor-manager/EditorManager';
9  import Guide from '../guide/Guide';
10 import Intro from '../intro/Intro';
11 import DebugButton from '../../buttons/debug/Debug';
12 import {SOURCE_CODE, TEST_CODE} from '../editor-manager/constants';
```

```
13 import {executeTestCase} from '../../../../packages/engine/Tester';
14 const Wrapped = (
15 code,
16 test,
17 testCaseTests,
18 sourceCodeTests,
19 guideContent,
20 whenDoneRedirectTo,
21 waitCodeToBeExecutedOnStep,
22 enableEditorOnStep,
23 trackSection,
24 testCaseStrategy,
25 sourceCodeStrategy,
26 disableEditor,
27 introContent,
28 enableIntroOnStep,
29 editorOptions,
30 attentionAnimationTo = []
31 ) => {
32 class Rocket extends Component {
33 state = {
34 code: {
35 [SOURCE_CODE]: code,
36 [TEST_CODE]: test
37 },
38 editorOptions: editorOptions || {
39 [SOURCE_CODE]: {
40 readOnly: true
41 },
42 [TEST_CODE]: {}
43 },
44 done: false,
45 showNext: false,
46 currentHint: 0,
47 initialStep: 0,
48 introEnabled: false,
49 intro: introContent || {
50 steps: [],
51 initialStep: 0
52 },
53 editorError: false,
54 }
```

```
55 }
56 }
57
```

El número de parámetros para probar la función es tan elevado que si alguien empezase a escribir un nuevo caso de prueba, probablemente olvidaría qué debe introducir para obtener el resultado deseado.

Farley (2021) muestra otro ejemplo de Jenkins, un proyecto *CI/CD* de código abierto. Describe un caso de prueba particular que construye un navegador para afirmar la URL que se está utilizando. En este ejemplo se podría haber conseguido el mismo resultado de una forma más sencilla utilizando un objeto regular que llame a un método para comprobarlo.

Aspectos a tener en cuenta

1. Revisar el SOLID para el caso de prueba y organizarlo en consecuencia (no temer refactorizar las pruebas).
2. ¿Estás añadiendo un caso de prueba que requiere cambiar demasiados elementos de la configuración? Reorganiza los *tests*.

El Inspector

Una prueba unitaria que viola la encapsulación para lograr una cobertura de código del 100%, pero que tiene tanta información sobre lo que ocurre en el objeto que cualquier intento de refactorización rompe la prueba existente y requiere que los cambios se reflejen en la prueba unitaria (Carr 2022).

El *Inspector* forma parte del Episodio 2 de la serie de videos[74] sobre los antipatrones de TDD, presentada por Codurance.

A menudo, el propósito de las pruebas se combina con la “inspección”, que puede resultar de acoplar el caso de prueba a detalles de implementación como *reflection* o exponer el comportamiento interno para inspeccionar la salida.

El proyecto Git Release Bot – Exponiendo detalles

A veces, al realizar pruebas, nos encontramos en situaciones en las que nos gustaría verificar que si hay un *input X* obtendremos el resultado Y. En el siguiente fragmento de código, aparecen los métodos *getFilesToWriteRelease* (línea 17) y *setFilesToWriteRelease* (línea 22), si pensamos en el contexto de la clase, ¿por qué existen estos métodos?

74 https://www.codurance.com/es/publications/tdd-y-anti-patrones-cap%C3%ADtulo-2

```
58 class Assembly
59 {
60 /* skipped code */
61 public function __construct(
62 FindVersion $findVersion,
63 FileRepository $fileRepository,
64 string $branchName,
65 FilesToReleaseRepository $filesToReleaseRepository
66 ) {
67 $this->findVersion = $findVersion;
68 $this->fileRepository = $fileRepository;
69 $this->branchName = $branchName;
70 $this->filesToReleaseRepository = $filesToReleaseRepository;
71 }
72 public function getFilesToWriteRelease(): array
73 {
74 return $this->filesToWriteRelease;
75 }
76 public function setFilesToWriteRelease(array $filesToWriteRelease)
77 {
78 $this->filesToWriteRelease = $filesToWriteRelease;
79 return $this;
80 }
81 public function packVersion(): Release
82 {
83 $filesToRelease = $this->getFilesToWriteRelease();
84 if (count($filesToRelease) === 0) {
85 throw new NoFilesToRelease();
86 }
87 $files = [];
88 /** @var File $file */
89 foreach ($filesToRelease as $file) {
90 $files[] = $this->fileRepository->findFile(
91 $this->findVersion->getProjectId(),
92 sprintf('%s%s', $file->getPath(), $file->getName()),
93 $this->branchName
94 );
95 }
96 $versionToRelease = $this->findVersion->versionToRelease();
97 $release = new Release();
98 $release->setProjectId($this->findVersion->getProjectId());
99 $release->setBranch($this->branchName);
```

```
100 $release->setVersion($versionToRelease);
101 $fileUpdater = new FilesUpdater($files, $release, $this-
    >filesToReleaseRepository);
102 $filesToRelease = $fileUpdater->makeRelease();
103 $release->setFiles($filesToRelease);
104 return $release;
105 }
106 }
107
```

En este sentido, si pensamos en la Programación Orientada a Objetos (McLaughlin, Pollice y West 2007), hablamos de invocar métodos en objetos. Dichas llamadas nos permiten interactuar y recibir un resultado. Pero, los métodos get/set rompen el encapsulamiento y existen únicamente con el propósito de realizar pruebas.

Inspeccionar el código con Reflection

Otra posible razón que nos hace experimentar a El *Inspector* es agregar complejidad al revisar una parte específica del código con *reflection*. Antes de continuar, recapitulemos qué es *reflection* y por qué se utiliza en primer lugar.

Baeldung[75] da algunos ejemplos de por qué puede ser necesario usar *reflection*. El uso principal es obtener una comprensión de cierta parte del código, métodos o propiedades, y actuar sobre ellos. El siguiente ejemplo lo muestra:

```
1  public class Employee {
2  private Integer id;
3  private String name;
4  }
5  @Test
6  public void whenNonPublicField_thenReflectionTestUtilsSetField() {
7  Employee employee = new Employee();
8  ReflectionTestUtils.setField(employee, “id”, 1);
9  assertTrue(employee.getId().equals(1));
10 }
```

Ahora que entendemos los superpoderes de *reflection*, ¿por qué podría perjudicar la testabilidad? Por suerte, hace unos años nos hicieron esta pregunta en *StackOverflow*.[76] El hilo es largo y muestra diferentes opiniones de

---

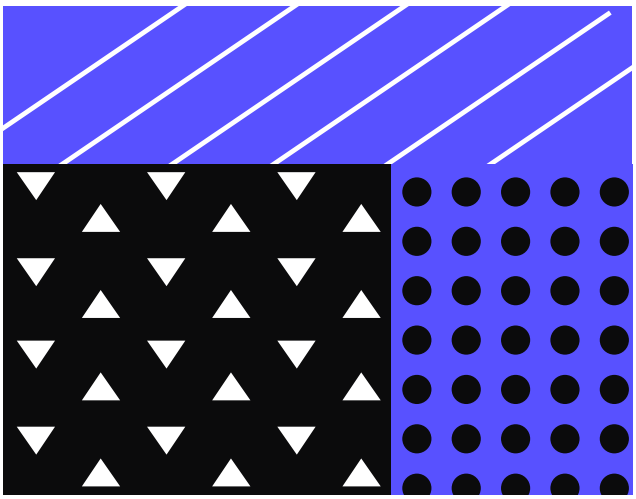
75 Guía de ReflectionTestUtils para pruebas unitarias.

76 ¿Es una mala práctica utilizar Reflection en las pruebas unitarias?

los desarrolladores. En resumen, se considera una mala práctica llegar a ese nivel de detalle al probar el código.

Aspectos a tener en cuenta

1. Evitar exponer métodos con el único propósito de inspeccionar la salida o definir entradas dentro de un caso de prueba.
2. Modificar el código de producción para poder testearlo es un smell de que hay algo mal en el diseño.



Nivel VI

Nivel IV

Este nivel muestra uno de los escenarios más difíciles que trae consigo el enfoque *test-first*. Vamos a ir a través de *El Mockery*, el más popular de los antipatrones según la encuesta.

- Cuidado con probar el *test* doble en lugar del código de producción.
- Un solo caso de prueba puede tener múltiples antipatrones a la vez.
- Evitar no limpiar los datos creados para un caso de prueba específico, se recomienda no compartir datos entre pruebas. Esto también se relaciona con *El Peeping Tom*.
- Evitar tener un test *suite* que tarde mucho tiempo en ejecutarse siempre que sea posible.

El Mockery

A veces hacer *mocking* puede ser bueno y útil. Pero, en ocasiones, los profesionales pueden perderse tratando de simular lo que no se está testeando. En este caso, una prueba unitaria contiene tantos *mocks*, *stubs y/o fakes* que el sistema bajo test ni siquiera se está probando, en su lugar, los datos devueltos por los *mocks* es lo que se está probando (Carr 2022).

El *Mockery* forma parte del Episodio 2 de la serie de videos[77] sobre los antipatrones de TDD, presentada por Codurance.

El *Mockery* es uno de los antipatrones más populares que experimentan los equipos de desarrollo. Parece que todo el mundo ha tenido alguna experiencia *mockeando* código para probar algún comportamiento. La primera idea de *mocking* es simple: evitar código externo para centrarse en lo que se quiere probar. Sin embargo, puede ser complicado de implementar correctamente.

El *mocking* también puede adoptar muchos enfoques diferentes. Por ejemplo, Fowler (2022b) y Bob (2022) clasificaron los distintos tipos de *mocking* del siguiente modo: *dummies, spies, fakes y stubs*. Aunque Uncle Bob se refiere a *True Mock*", mientras que Martin Fowler se refiere a *Test Double*".

Uncle Bob explica que el término mocks se popularizó porque es más fácil decir: "Voy a hacer mock de eso" o "¿puedes mockear esto?".

La diferencia entre ellos es importante porque nosotros, como personas que desarrollamos código, tenemos la costumbre de llamar a todo mock, causando así confusión sobre la verdadera intención. Por ejemplo, Spring.io va incluso más

77 https://www.codurance.com/es/publications/tdd-y-anti-patrones-cap%C3%ADtulo-2

allá, evitando el debate sobre si debemos usar mocks o no (refiriéndose a la Escuela Clásica y la Escuela de Londres de TDD)[78] – Yo haré lo mismo, pero eso me da que pensar que este es un tema que merece una entrada de blog.

Martin Fowler también escribió sobre este mismo tema (Fowler 2022a) y coincide en que durante un tiempo, él también pensó que mocks y stubs eran lo mismo.[79]

- DUMMIES – Pasas algo y no te importa a quién lo utiliza, a menudo, el objeto no se utiliza en absoluto.
- STUB – A diferencia de los dummies, los *stubs* son objetos creados para que te importe cómo se utilizan. Por ejemplo, para engañar a una autorización y probar si el usuario puede o no realizar ciertas acciones en el sistema.
- SPIES – Para constatar que un método fue llamado por el sistema bajo prueba, como menciona un post de Martin (2014): "Puedes usar *Spies* para ver dentro del funcionamiento de los algoritmos que estás testeando".
- TRUE MOCKS – Se interesa por el comportamiento en lugar del retorno de las funciones. Le importa saber qué funciones se invocan, con qué argumentos y con qué frecuencia.
- FAKES – Los *Fakes* tienen lógica de negocio, por lo que pueden manejar el sistema bajo prueba con diferentes conjuntos de datos.

El antipatrón Mockery se basa en la misma definición enunciada por Uncle Bob, que hace referencia a todos los mocks. Para aclarar las cosas, dividamos el antipatrón en dos partes: la primera es el número excesivo de mocks necesarios para probar una clase. Por ejemplo, también está relacionado con El Excessive Setup .

La segunda parte consiste en probar el mock en lugar de su interacción con el código bajo prueba. Por ejemplo, si tenemos el siguiente fragmento de código que muestra una interacción con una plataforma de pago.

```
1  /**
2  * Two constructor dependencies, both need to be
3  * mocked in order to test the process method.
4  */
5  class PaymentService(
6  private val userRepository: UserRepository,
7  private val paymentGateway: PaymentGateway
8  ) {
9  fun process(
10 user: User,
```

78 Are You Chicago Or London When It Comes To TDD? – https://www.youtube.com/watch?v=_S5iUf0ANyQ.

79 En la práctica, los profesionales siguen tratando los *test dobles* como si fueran iguales. La comunicación se reduce a "haz mock esto " o "haz mock de aquello".

```
paymentDetails: PaymentDetails
): Boolean {
if (userRepository.exists(user)) {
return paymentGateway.pay(paymentDetails)
}
return false
}
}
With the given test:
class TestPaymentService {
private val userRepository: UserRepository = mockk()
private val paymentGateway: PaymentGateway = mockk()
private val paymentService = PaymentService(
userRepository,
paymentGateway
)
@Test
fun paymentServiceProcessPaymentForUser() {
val user: User = User()
every { userRepository.exists(any()) } returns true
every { paymentGateway.pay(any()) } returns true // setting up the return for the mock
assertTrue(paymentService.process(user, PaymentDetails())) // asserting the mock
}
}
```

Aspectos a tener en cuenta

1. ¿Razones históricas/estilos de TDD?
2. Por muy sencillo que sea el código, lo más fácil es abusar de él
3. No tener experiencia en TDD

El One

Una combinación de varios patrones, en particular *The Free Ride* y *The Giant*, un test unitario que contiene sólo un método de prueba que evalúa todo el conjunto de funcionalidades que tiene un objeto. Un indicador común es que el método de prueba suele ser el mismo que el nombre del test unitario, y contiene múltiples líneas de configuración y aserciones (Carr 2022).

El One forma parte del Episodio 6 de la serie de videos[80] sobre los antipatrones de TDD, presentada por Codurance.

A pesar de su nombre, El One es el antipatrón que combina diferentes antipatrones. Por definición, está relacionado con *El Giant* y *El Free Ride*.

Como ya hemos visto, *El Giant* aparece cuando un *test* intenta hacerlo todo a la vez dentro de un mismo caso de prueba .

Existen algunas variantes. El siguiente fragmento fue extraído del libro *xUnit* (Meszaros 2007) y representa el antipatrón *Giant* (aunque el número de líneas no es tan elevado como en el primer episodio). A pesar de ser más corto en longitud, sigue siendo un buen ejemplo porque el caso de prueba está tratando de ejercitar todos los métodos dentro del objeto *Flight* en un solo test:

```
1  public void testFlightMileage_asKm2() throws Exception {
2  // set up fixture
3  // exercise constructor
4  Flight newFlight = new Flight(validFlightNumber);
5  // verify constructed object
6  assertEquals(validFlightNumber, newFlight.number);
7  assertEquals(“”, newFlight.airlineCode);
8  assertNull(newFlight.airline);
9  // set up mileage
10 newFlight.setMileage(1122);
11 // exercise mileage translator
12 int actualKilometres = newFlight.getMileageAsKm();
13 // verify results
14 int expectedKilometres = 1810;
15 assertEquals( expectedKilometres, actualKilometres);
16 // now try it with a canceled flight
17 newFlight.cancel();
18 try {
19 newFlight.getMileageAsKm();
20 fail(“Expected exception”);
21 } catch (InvalidRequestException e) {
22 assertEquals( “Cannot get cancelled flight mileage”,
23 e.getMessage());
24 }
25 }
```

80 https://www.codurance.com/es/publications/tdd-and-anti-patterns-capitulo-6

Los comentarios nos dan incluso una pista sobre cómo dividir el caso de prueba único en varios tests. Del mismo modo, en este ejemplo también se puede observar El Free Ride, ya que para cada configuración aparecen algunas aserciones.

El proyecto Jenkins

En el siguiente fragmento puede ser más sencillo de detectar cuando aparece el *Free Ride*. Como ya se ha mostrado en con anterioridad este ejemplo se ha extraído del repositorio *Jenkins*. En este caso, la distinción entre El *Free Ride* y El *Giant* es algo borrosa, pero aún así, es fácil evitar que un solo caso de prueba haga demasiado.

```
1  public class ToolLocationTest {
2  @Rule
3  public JenkinsRule j = new JenkinsRule();
4  @Test
5  public void toolCompatibility() {
6  Maven.MavenInstallation[] maven = j.jenkins.getDescriptorByType(Maven.
   DescriptorImpl.class).getInstallations();
7  assertEquals(1, maven.length);
8  assertEquals("bar", maven[0].getHome());
9  assertEquals("Maven 1", maven[0].getName());
10 Ant.AntInstallation[] ant = j.jenkins.getDescriptorByType(Ant.DescriptorImpl.
   class).getInstallations();
11 assertEquals(1, ant.length);
12 assertEquals("foo", ant[0].getHome());
13 assertEquals("Ant 1", ant[0].getName());
14 JDK[] jdk = j.jenkins.getDescriptorByType(JDK.DescriptorImpl.class).
   getInstallations();
15 assertEquals(Arrays.asList(jdk), j.jenkins.getJDKs());
16 assertEquals(2, jdk.length); // JenkinsRule adds a 'default' JDK
17 assertEquals("default", jdk[1].getName()); // make sure it's really that we're
   seeing
18 assertEquals("FOOBAR", jdk[0].getHome());
19 assertEquals("FOOBAR", jdk[0].getJavaHome());
20 assertEquals("1.6", jdk[0].getName());
21 }
22 }
23
```

El Generous Leftovers

Una instancia en la que un test crea datos que se mantienen en algún lugar, y otro *test* reutiliza los datos para sus propios fines. Si el "generador" no se ejecuta, o lo hace después del segundo test, la prueba que utiliza esos datos fallará rotundamente (Carr 2022).

El *Generous Leftovers* forma parte del Episodio 2 de la serie de videos[81] sobre los antipatrones de TDD, presentada por Codurance.

Al aplicar TDD, tareas como configurar el estado en el que se ejecutará la prueba forma parte de los fundamentos básicos, ya sea mediante la configuración de *fake data, listeners, authentication* o cualquier otro estado dependiente.

Los definimos porque son cruciales para el test, pero a veces nos olvidamos de restablecer el estado a como estaba antes de ejecutar el *test*. En resumen, las fases que debe tener un caso de prueba, según Fowler (2022) y Meszaros (2007), son: *setup, exercise, verify, teardown.* El restablecimiento del estado debe producirse dentro de la fase de desmontaje para evitar que afecte a otros casos de prueba.

Si no se restablece el estado, pueden producirse diferentes problemas. El primero es hacer que falle la siguiente prueba, que de otro modo habría pasado sin problemas. La siguiente lista trata de describir algunos escenarios en los que esto puede ocurrir.

1. Establecer listeners y olvidarse de eliminarlos también puede causar fugas de memoria.
2. Poblar datos sin eliminarlos - cosas como archivos, bases de datos o incluso caché.
3. Depender de una prueba para crear los datos necesarios y utilizarlos en otra prueba.
4. Limpiar los test dobles.[82]

Si pensamos en el tercer escenario de la lista anterior, este comportamiento podría ser algo complicado a la hora de realizar tests. Por ejemplo, el uso de datos persistentes es imprescindible para las pruebas *end-to-end*. Por otro lado, el último escenario de la lista es una fuente común de errores durante el desarrollo guiado por tests. A menudo, como el *mock* se suele utilizar para recoger llamadas en el objeto (y verificarlo más tarde), es común olvidarse de restaurar su estado.

---

81 https://www.codurance.com/es/publications/tdd-y-anti-patrones-cap%C3%ADtulo-2

82 Relacionado con mantener el estado entre pruebas como se discute anteriormente.

Codingwithhugo[83] demuestra este comportamiento en un fragmento de código usando Jest, dando el siguiente caso de prueba (y asumiendo que están en el mismo ámbito):

```
1  const mockFn = jest.fn(); // setting up the mock
2  function fnUnderTest(args1) {
3  mockFn(args1);
4  }
5  test('Testing once', () => {
6  fnUnderTest('first-call');
7  expect(mockFn).toHaveBeenCalledWith('first-call');
8  expect(mockFn).toHaveBeenCalledTimes(1);
9  });
10 test('Testing twice', () => {
11 fnUnderTest('second-call');
12 expect(mockFn).toHaveBeenCalledWith('second-call');
13 expect(mockFn).toHaveBeenCalledTimes(1);
14 });
```

El primer test que llame a la función bajo prueba pasará, pero el segundo fallará. La razón de esto es la ausencia de un restablecimiento del *mock*. La prueba falla al decir que el *mockFn* fue llamado dos veces. Corregir esto es tan fácil como:

```
1  test('Testing twice', () => {
2  mockFn.mockClear(); // clears the previous execution
3  fnUnderTest('second-call');
4  expect(mockFn).toHaveBeenCalledWith('second-call');
5  expect(mockFn).toHaveBeenCalledTimes(1);
6  });
```

Los profesionales de *JavaScript* no suelen enfrentarse a este tipo de comportamientos, ya que el ámbito funcional de *JavaScript* evita que se produzcan de forma automática dentro de las construcciones del lenguaje. Sin embargo, tener en cuenta estos detalles puede marcar la diferencia a la hora de escribir pruebas.[84]

Aspectos a tener en cuenta

5. Falta de práctica en TDD.
6. Los *fixtures* persistentes pueden convertirse en una fuente de errores.

---

83 *Jest set, clear and reset mock/spy/stub implementation.*

84 Marvin Hagemeister escribió una entrada en su blog titulada *Ejecutar 1000 tests en un 1s* y gran parte del rendimiento obtenido se debió a desactivar algún comportamiento predeterminado de limpieza de recursos que tiene Jest. Cuando se necesita tal nivel de optimización, tener control sobre el estado se vuelve crucial.

7. Los tests están acoplados a una secuencia específica por ejecutar.
8. Mix de diferentes tipos de pruebas, *integration/unit/end-to-end*.

El Slow Poke

Una prueba unitaria que se ejecuta con mucha lentitud. Cuando la ponemos en marcha, te da tiempo de ir al baño, a hacer un té o, peor aún, terminar la prueba antes de irse a casa al final del día (Carr 2022).

El *Slow Poke* me recuerda al videojuego japonés Pokémon. El juego contiene una criatura con este nombre que es capaz de hacer que sus oponentes sean menos eficientes a la hora de atacar. Esta característica la comparte con El *Slow Poke*. En este caso, sin embargo, este antipatrón reduce la eficacia de la ejecución de un conjunto de pruebas. Por lo general, hace que un conjunto de tests automatizados tarde más tiempo en ejecutarse y, como resultado, alarga el ciclo de retroalimentación.

El código relacionado con el tiempo suele ser difícil de manejar en un caso de prueba y, a menudo, es una razón por la que aparece El *Slow Poke*. Este tipo de código requiere que manipulemos el tiempo de diferentes maneras para permitirnos simular escenarios basados en él. Por ejemplo, podríamos querer probar que una ejecución de pago automatizada se activa al final de un mes dentro de un sistema de pago.

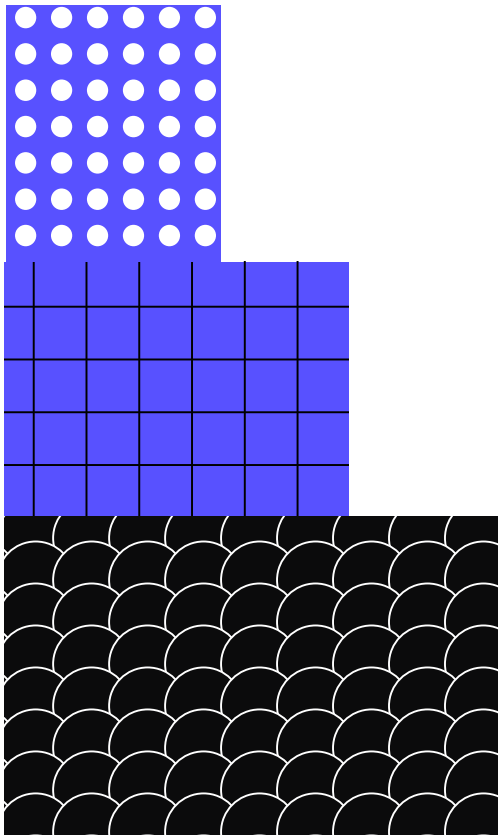
Para ello, necesitamos manejar la hora y comprobar una fecha concreta (el último día del mes, en este caso). Junto con la fecha, probablemente también tendríamos que simular la hora (tal vez alrededor de la mañana o la tarde). Es decir, necesitamos una manera de gestionar el tiempo sin esperar hasta el final del mes para ejecutar la prueba.

El tiempo es asíncrono y si dependemos directamente de él, conduce a la falta de determinación, como menciona Fowler (2011). Afortunadamente, hay una manera de superar esta situación, pero utilizando *mocking* en nuestros tests.

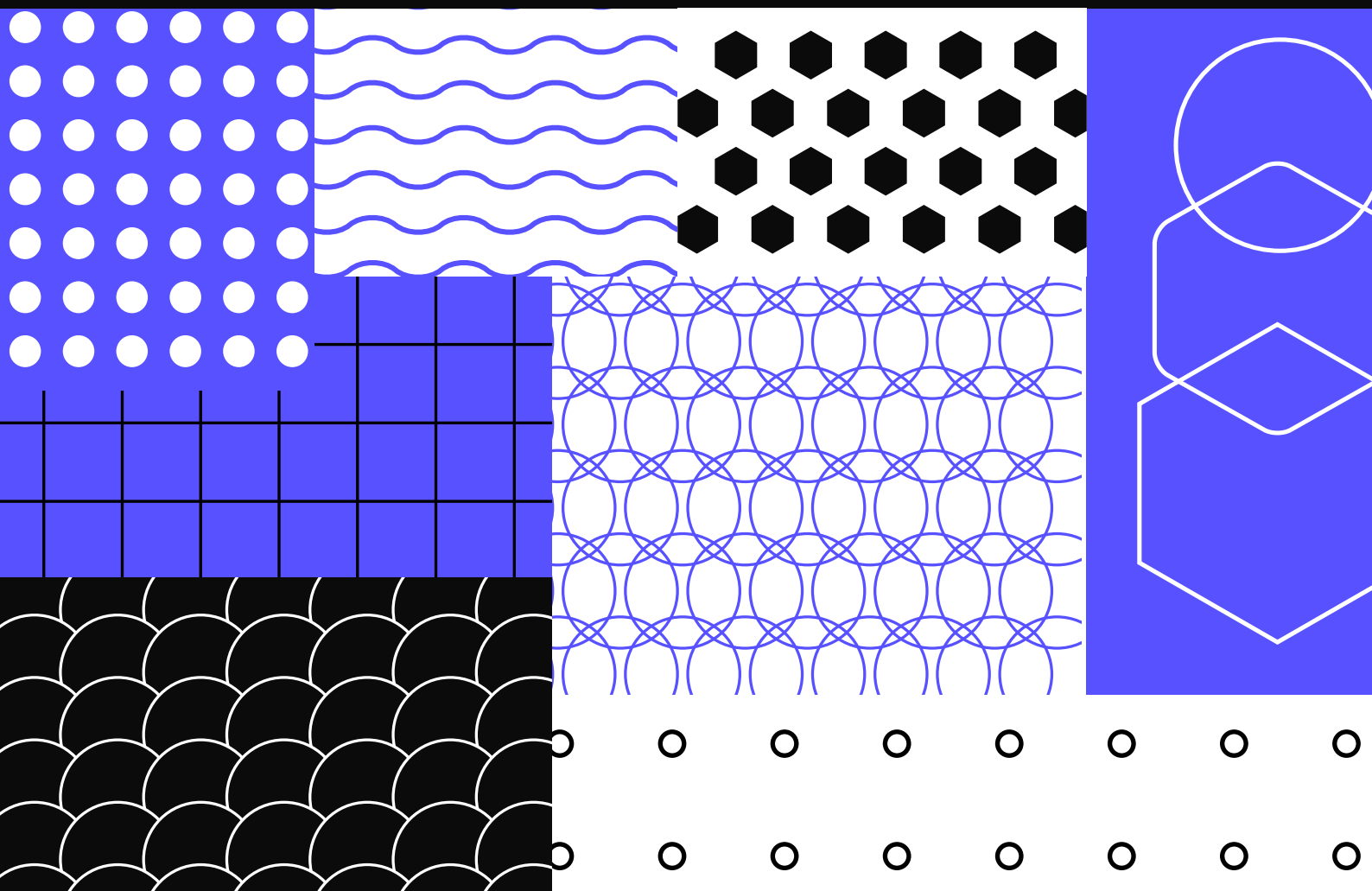
Empujar hacia más pruebas de integración o *tests end-to-end* también puede transformar el conjunto de pruebas en una tarea pesada, tardando largas horas o incluso días en completarse. Este enfoque también está relacionado con la estrategia de pruebas del cono de helado. En un escenario ideal, seguirías una estrategia de pruebas piramidal en la que tendrías como base más pruebas unitarias, unas pocas pruebas de integración y muy pocas pruebas *end-to-end* (Vocke 2018).

Aspectos a tener en cuenta

- Bases de código monolíticas que requieren más tiempo de compilación que de ejecución de las pruebas.
- Testear con estrategias que el *framework* ya proporciona.
- Tener demasiadas integraciones o pruebas e2e en lugar de unitarias (código con forma de helado visto anteriormente)



Conclusiones – Patrones que dificultan el TDD



Conclusiones – Patrones que dificultan el TDD

Si has llegado al final de este libro, deberías conocer y comprender mejor algunos de los patrones que pueden dificultar las pruebas de código.

Hemos hablado sobre las razones que nos llevaron a la creación de este libro, empezando por los retos a los que se enfrentan los desarrolladores a diario a la hora de testear aplicaciones. También cubrimos la *Test Pyramid* que se utiliza comúnmente hoy en día en la industria del desarrollo de software. Además, vimos que diferentes autores dieron a conocer los antipatrones enumerados de diferentes formas.

Hemos explorado lo que se comenta sobre los antipatrones en la actualidad.

En una encuesta se preguntó a desarrolladores qué sabían sobre los antipatrones de TDD y cuál era el posible impacto de estos antipatrones en su codificación diaria. Los datos recabados de los profesionales se agruparon en cuatro categorías diferentes:

- Experiencia profesional – conoce mejor quién responde el sondeo..
- Prácticas de TDD en el día a día
- Prácticas de TDD en empresas en las que he trabajado - prácticas que siguen las compañías con respecto a TDD.
- Antipatrones – sección dedicada a obtener información si los desarrolladores son capaces de recordar los antipatrones.
- Datos personales - correo electrónico para compartir los resultados.

Los datos de la encuesta trajeron consigo debates sobre cómo se aplica TDD en las organizaciones.

Con la introducción y los datos recopilados, pasamos a profundizar en la lista de antipatrones.

El método utilizado en este libro para presentar los antipatrones es diferente al que se suele encontrar en Internet. Aquí adoptamos el concepto de niveles que representan el viaje que un practicante va a recorrer mientras aprende TDD y los antipatrones que pueden aparecer durante ese trayecto.

En total, hemos dividido los antipatrones en cuatro niveles:

Nivel 1 – Cubre la mayoría de los antipatrones tal y como se detectan por primera vez cuando los profesionales están aprendiendo a testear código.

Nivel 2 – Muestra el camino de un principiante hacia un nivel intermedio y los retos a los que se enfrenta durante la práctica de TDD. Además, observamos que los antipatrones pueden estar relacionados con la falta de uso de principios de desarrollo de software, como la Single Responsability en los casos de prueba.

Nivel 3 – Es una continuación del nivel anterior ya que profundiza en el diseño de software y relaciona la calistenia de objetos con los antipatrones presentados.

Nivel 4 – Abordamos antipatrones como El Mockery, que introduce una nueva forma de pensar sobre cómo probar código utilizando test doubles y elementos que ralentizan el conjunto de pruebas, también conocido como El Slow Poke.

Los niveles se crearon para dar a este libro una estructura, de manera que los antipatrones puedan emparejarse con diferentes temas mientras se practica TDD. Esto no significa que sólo los principiantes se enfrentarán a los escenarios representados en el nivel I o que sólo los desarrolladores experimentados se enfrentarán a los escenarios del nivel IV.

Con este contenido, espero que podamos llegar a un acuerdo sobre los patrones que dificultan el testing.

Aunque se denominan “antipatrones”, la intención es convertirlos en “patrones” que dificultan las pruebas de código y quitarles el tono negativo. Ya que la mayoría de las bases de código pueden tener al menos uno o más de ellos y los profesionales se enfrentarán con varios en su camino.

Lo que nos enseña la experiencia

No recuerdo quién me dió a conocer esta obra, pero recuerdo haber leído el emblemático libro de Kent Beck, *TDD by Example*, durante mis trayectos al trabajo. Cada página que pasaba era un descubrimiento. Esas páginas también entrañaban cierta frustración. El enfoque de *baby steps* entraba en conflicto con la confianza que tenía en que el código iba a funcionar. “Pasará la prueba, está fuertemente codificado”. Uno de los conceptos erróneos enumerados por Olena Borzenko en su charla del 2021 en la conferencia TDD estaba relacionado con ese tipo de pensamiento.

Uno de los mayores retos para los desarrolladores es seguir adelante con el enfoque test-first, independientemente del entorno y del equipo en el que trabajemos. Estamos aprendiendo constantemente y la adopción de prácticas que se dan por buenas por defecto no es habitual para la mayoría de nosotros.

Por otro lado, he reflexionado sobre mis primeros pasos en el mundo del desarrollo y cómo el contexto en el que me encontraba no me ayudó a mejorar mis habilidades técnicas, concretamente en TDD, lo que también conecta con los resultados que vimos en la encuesta más del 50% de las empresas no practican TDD).

Recuerdo ser parte de proyectos que no tenían una cultura de *test-first*, en cambio el enfoque era construir desde cero, y en eso, no estoy solo. (Pérez 2022).

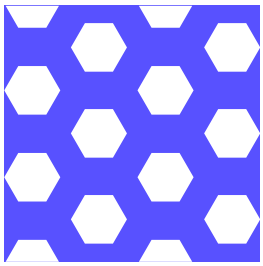
Hacia dónde ir

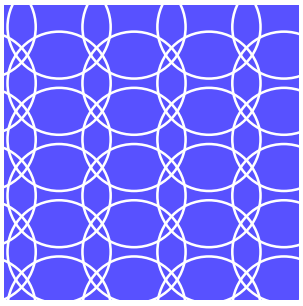
A pesar del alcance de este libro, es posible que también quieras profundizar un poco más en la práctica de TDD y en lo que otros autores opinan, por ello he recopilado la siguiente lista de recursos que podrían ayudarte.

- YouTube tiene muchas charlas en torno a TDD, para mantener un registro de lo que me pareció interesante he creado una playlist que puede valer la pena revisar. El objetivo era crear una lista de reproducción progresiva, empezando por lo básico e incrementando la complejidad gradualmente.
- *Effective Software Testing* por Mauricio Aniche – Este libro es una lectura obligada para los desarrolladores, ya que comparte un punto de vista detallado sobre cómo escribir tests y al mismo tiempo ofrece una guía a seguir.
- No hemos hablado de la escuela de la London school x Chicago school de TDD y merece la pena echarle un vistazo también, ya que podría influir en los patrones a los que probablemente te enfrentarás dependiendo del enfoque que elijas.

A lo largo de este libro hemos citado varios libros que cubren diferentes aspectos de TDD, peero aún no hemos hecho referencia a cómo mantenerse al día con la práctica más allá de la literatura.

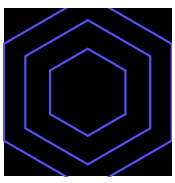
Aunque los libros cubren temas específicos con gran detalle, he descubierto que interactuar con distintas comunidades de desarrollo de software es el mejor mecanismo para conocer nuevos enfoques e ideas para mejorar la práctica de TDD.





Hoy en día hay un gran número de comunidades de desarrollo de software en todo el mundo. La mayoría pueden encontrarse en meetup.com. Por ejemplo, Codurance organiza eventos periódicos y Tech Excellence también organiza sesiones centradas en competencias técnicas.

Entrar en contacto con gente de comunidades de software no sólo te ayuda a aprender nuevas ideas y a reforzar tus conocimientos técnicos, sino que también es una oportunidad para que compartas tus conocimientos, dudas y experiencias. A pesar de la corta lista de grupos que he mencionado aquí, seguro que encuentras una comunidad cerca de ti que trate tus temas favoritos.



Anexo

Aquí encontrarás recursos relacionados con el libro.

Google form – Survey

Aniche, Mauricio. 2022. “Mauricio Aniche: How Code Coverage Can Be Used and Abused to Guide Testing.”
https://www.youtube.com/watch?v=f1DueZcSxRse.com/watch?v=f1DueZcSxRs

Beck, Kent. 2003. *Test-Driven Development: By Example.* Addison-Wesley Professional.

Bob), Robert C. Martin (Uncle. 2022. “The Little Mocker.”
https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html

Borzenko, Olena. 2021. “TDD Misconceptions.”
https://www.youtube.com/watch?v=TbkBMeAt4K0

Bugayenko, Yegor. 2021. “SSD 14/16: Test Patterns and Anti-Patterns.”
https://www.youtube.com/watch?v=KiUb6eCGHEY

Carr, James. 2022. “TDD Anti-Patterns.”
https://web.archive.org/web/20100105084725/http://blog.james-carr.org/2006/11/03/tdd-anti-patterns

Cohn, Mike. 2009. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional.

Dijkstra, Edsger Wybe, and others. 1970. “Notes on Structured Programming.” Technological University, Department of Mathematics.

Farley, Dave. 2021. “When Test Driven Development Goes Wrong.”
https://www.youtube.com/watch?v=UWtEVKVPBQ0&feature

Fowler, Martin. 2011. “Eradicating Non-Determinism in Tests.”
https://martinfowler.com/articles/nonDeterminism.html

———. 2022a. “Mocks Aren’t Stubs.”
https://martinfowler.com/articles/mocksArentStubs.html

———. 2022b. “TestDouble.”
https://martinfowler.com/bliki/TestDouble.html

Freeman, Steve, and Nat Pryce. 2009. *Growing Object-Oriented Software, Guided by Tests*. Pearson Education.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Pearson Education.
https://books.google.es/books?id=6oHuKQe3TjQC

Glitzel, Rodaney. 2022. “Singleton and Unit Testing.”
https://stackoverflow.com/questions/8256989/singleton-and-unit-testing/8263599#8263599

Hermans, Felienne. 2021. *The Programmer’s Brain: What Every Programmer Needs to Know About Cognition.* Simon; Schuster.

jestjs.io. 2021. “Testing Asynchronous Code.”
https://jestjs.io/docs/asynchronous

Mancuso, Sandro. 2018. “DevTernity 2018: Sandro Mancuso – Does TDD Really Lead to Good Design?”
<https://www.youtube.com/watch?v=KyFVA4Spcgg

Marabesi, Matheus. 2022. “Improving Testing Assertions.”
https://www.codurance.com/publications/improving-testing-assertionsnce.com

Marabesi, M, and I Frango Silveira. 2020. “EVALUATION of Testable, a Gamified Tool to Improve Unit Test Teaching.” In *INTED2020 Proceedings,* 330–38. IATED.

Martin, Robert C. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.

Martin, Robert C. 2014. “The Little Mocker.”
https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker

Martin, Robert Cecil. 2017. “Clean Architecture a Craftsman’s Guide to Software Structure and Design.”
https://archive.org/details/CleanArchitecture/page/n179/mode/2up

McLaughlin, Brett, Gary Pollice, and David West. 2007. *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to Ooa&D*. “O’Reilly Media, Inc.”.

Meszaros, Gerard. 2007. *XUnit Test Patterns: Refactoring Test Code*. Pearson Education.

Okken, Brian. 2022. *Python Testing with Pytest*. Pragmatic Bookshelf.

Percival, Harry, and Bob Gregory. 2020. *Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices*. “O’Reilly Media, Inc.”.

Pérez, Julio César. 2022. “Una Historia de Testing.”
https://www.meetup.com/pt-BR/codurance-craft-events/events/287021364/.com

Radziwill, Nicole. 2020. “Accelerate: Building and Scaling High Performance Technology Organizations. (Book Review) 2018 Forsgren, N., J. Humble and G. Kim. Portland or: IT Revolution Publishing. 257 Pages.” Taylor & Francis.

Steinberg, Daniel H. 2008. *The Thoughtworks Anthology: Essays on Software Technology and Innovation*. Pragmatic Bookshelf.

Stemmler, Khalil. 2022. *SOLID – Introduction to Software Design and Architecture with Typescript*.
https://solidbook.io.

Vocke, Ham. 2018. “The Practical Test Pyramid.”
https://martinfowler.com/articles/practical-test-pyramid.html

Wang, Yuqing, Maaret Pyhäjärvi, and Mika V. Mäntylä. 2020. “Test Automation Process Improvement in a Devops Team: Experience Report.” In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 314–21. https://doi.org/10.1109/ICSTW50294.2020.00057.

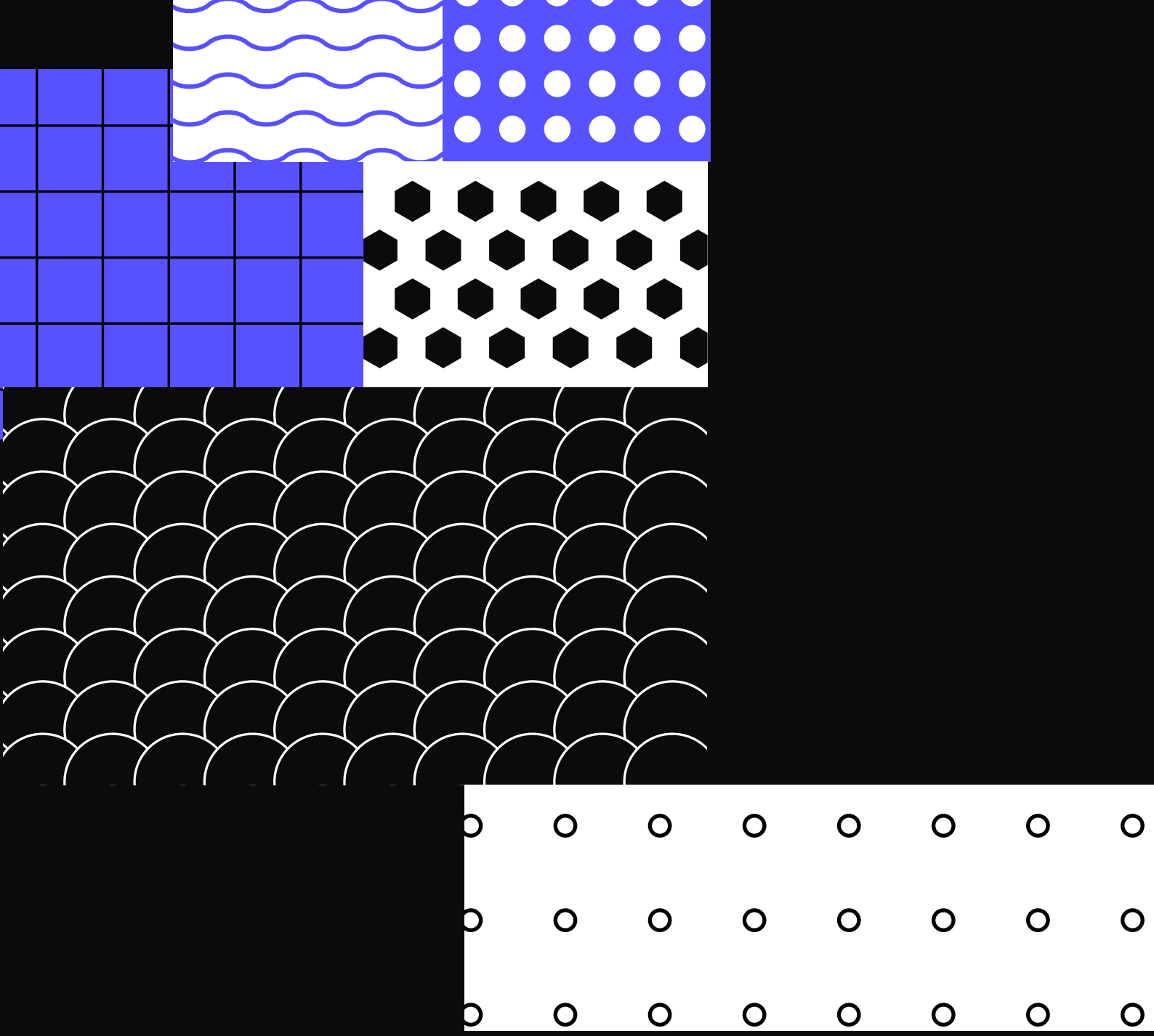
Sobre el autor

Matheus Marabesi es ingeniero informático y un absoluto adicto a todo lo que tiene que ver con el desarrollo de software en general. Su pasión por la profesión es lo que le impulsa a querer compartir lo que mejor sabe hacer. Es un software craftsperson en toda regla. Además de ser un aspirante a investigador centrado en el testing de software y la gamificación, ha desarrollado Testable, una herramienta gamificada que pretende cambiar la percepción que tienen muchos profesionales del desarrollo de software sobre que aprender testing es algo aburrido.

Desde muy pequeño empezó con su primer ordenador y el famoso Windows XP, mientras lo compaginaba con sus clases de dibujo; pero cuanto más tiempo pasaba con su ordenador, más le interesaba y el dibujo se quedó atrás. Así que empezó a asistir a cursos de formación en hardware y redes informáticas, pero su interés seguía creciendo y ampliándose a todo tipo de tecnologías y todo lo que tenía que ver con la computación, así que con ese deseo profundo obtuvo su primera titulación como ingeniero.

Actualmente Matheus forma parte del equipo de Software Craftsperson de Codurance Spain. Podéis compartir ideas e intereses con él en muchos de los Meetups que organizamos para la comunidad, y lo veréis participando activamente en Coding Dojos, workshops y muchos de los contenidos que nos gusta compartir con nuestros seguidores. Además está muy implicado en proyectos de código abierto, foros en los que se trabajen buenas prácticas relacionadas con aplicaciones bien diseñadas, código limpio, testing y gamificación. Si os interesa echar un vistazo a sus aportaciones más profesionales aquí tenéis su blog. https://marabesi.com.

En su tiempo libre le gusta jugar con Raspberry PI, sumergirse en el mundo IoT y salir a correr con una buena playlist de música electrónica en sus auriculares.



codurance

hello@codurance.com

codurance.com