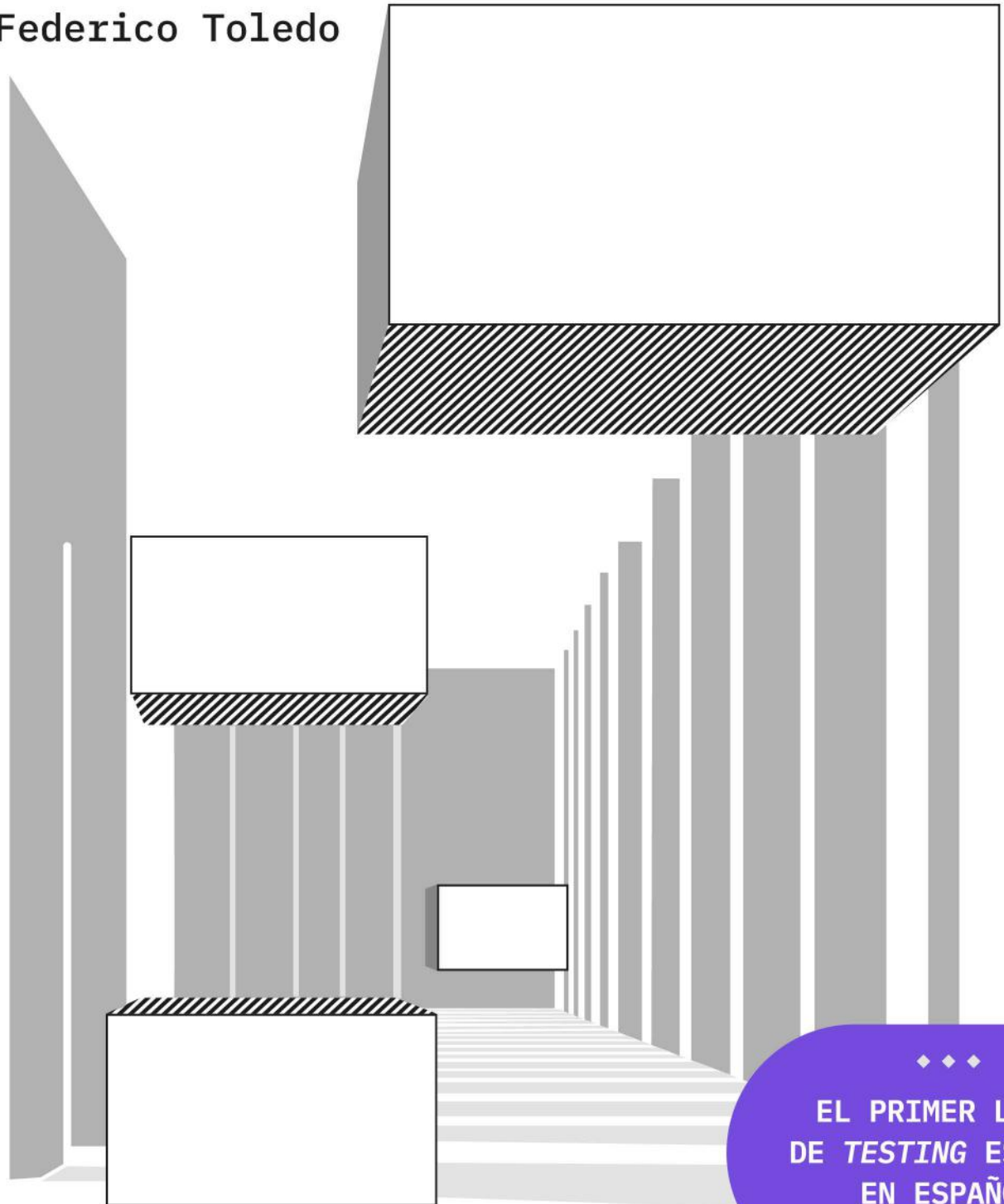


Introducción a las pruebas de sistemas de información

Federico Toledo



◆ ◆ ◆
EL PRIMER LIBRO
DE *TESTING* ESCRITO
EN ESPAÑOL

TERCERA EDICIÓN

INTRODUCCIÓN A LAS PRUEBAS DE SISTEMAS DE INFORMACIÓN

Federico Toledo

*** El primer libro de *testing* ***
escrito en español

Abstracta, Montevideo, Uruguay, 2024

Sobre el autor

Federico ha dedicado toda su carrera a la calidad de *software*. Se define como un *quality engineer* o *quality advocate*, dado que siempre se encuentra buscando formas de promover la calidad y el *testing* en los ciclos de desarrollo, de modo continuo.

Es ingeniero en computación por la Universidad de la República (Udelar) en Uruguay, y doctor en informática por la Universidad de Castilla-La Mancha (UCLM), España. Desde 2005, ha trabajado en diversas temáticas relacionadas al *testing* y la calidad de *software*.

En 2008, cofundó Abstracta en Uruguay, una empresa líder en soluciones tecnológicas, especializada en *testing*, desarrollo de *software* y soluciones impulsadas por inteligencia artificial (IA).

Actualmente, la compañía cuenta con sedes en Estados Unidos, el Reino Unido, Chile, Colombia y Uruguay. Su portafolio incluye herramientas como GXtest, Apptim (antes conocida como Monkop), PrivateGPT, JMeter DSL, Citizen Developer Copilot y más. Además, ofrece oportunidades de aprendizaje mediante su plataforma Abstracta Academy.

Federico también cofundó el proyecto social y educativo Nahual Uruguay y la conferencia pionera de *testing* de América Latina llamada TestingUy.

Conferencista internacional sobre prácticas de *testing* y calidad de *software* y autor de artículos en múltiples blogs, ha colaborado con investigación y docencia en *testing* en universidades como Udelar, Universidad Católica y Universidad de la Empresa en Uruguay, UCLM en España y el Consejo Nacional de Investigación (CNR, por su sigla en el idioma de origen), en Italia.

En 2014, publicó la primera edición de *Introducción a las pruebas de sistemas de información*, el primer libro práctico de *testing* en español, que se halla disponible de manera gratuita, con el fin de extender el acceso al conocimiento.

En 2020, creó su *podcast Quality sense*, en el cual desde entonces realiza entrevistas a referentes en el área periódicamente, a favor del fortalecimiento de la comunidad de tecnología de la información (TI). En 2021, se graduó de Stanford + LBAN SLEI, un programa dedicado a personas emprendedoras de América Latina que buscan expandir sus empresas.

En 2022, logró que uno de los eventos más reconocidos en *Testing de Performance* se llevara a cabo en Uruguay, el WOPR (Workshop on Performance and Reliability). El mismo año, creó el evento de *software* de calidad Quality Sense Conf, en el que participaron más de 15 referentes internacionales.

Desde entonces, su pasión por el intercambio de conocimientos siguió creciendo cada vez más. En 2023, fundó WOPR Latam, la primera edición del prestigioso evento realizada en español. El mismo año, la Quality Sense Conf comenzó su gira por la región y llegó a Chile. ¿Su más cercano destino?, Colombia en 2024, y el plan hacia adelante ya se encuentra en marcha.

Federico nació en Montevideo, Uruguay, y se crió en una zona rural en las afueras de Atlántida. Cuando comenzó a estudiar, viajaba cuatro horas por día para hacerlo posible, en trayectos que estaban divididos en tramos en bicicleta, ómnibus y caminatas.

A los 18 años, con gran esfuerzo y apoyo de su mamá, se mudó a Montevideo para estudiar. Gracias a su perseverancia y dedicación, logró obtener una beca de la Agencia Nacional de Investigación e Innovación (ANII) y realizó su doctorado en España. Vivió tres años en Ciudad Real, España, y tres meses en Pisa, Italia, como parte de sus estudios.

Cuando terminó su doctorado, decidió recolectar material que había elaborado a lo largo de su carrera y desarrollar nuevos tópicos, para escribir un libro y ofrecerlo de forma gratuita, con el fin de que más personas pudieran dedicarse al *testing* en Uruguay y crear una comunidad.

Su desarrollo profesional y personal lo llevó a residir tres años en Berkeley, California, Estados Unidos, junto con su esposa, Alejandra. A mitad de 2022, regresaron a Uruguay, y actualmente viven en El Pinar con su beba, Sofía.

PRÓLOGO

Para el prólogo de esta edición, les pedimos a varios amigos y referentes de la industria que nos dieran su testimonio. Nos llenó de alegría y agradecimiento leer sus palabras, que aquí compartimos con ustedes:

Introducción a las pruebas de sistemas de información, de Federico Toledo, es una obra pionera en español que aborda de manera integral y práctica los diversos tipos de *testing*. El libro es una guía completa que cubre desde conceptos básicos hasta técnicas avanzadas de diseño de pruebas funcionales, automatizadas y de *performance*, así como las habilidades esenciales para los *testers*. Con un enfoque dinámico y práctico, proporciona herramientas y metodologías efectivas para enfrentar los desafíos del *testing* en el entorno profesional, destacando la importancia de integrar aspectos técnicos y humanos en el proceso de *testing*. Federico Toledo combina su destacada trayectoria en el campo del *testing* con un estilo claro y accesible, facilitando la rápida incorporación de conocimientos para el lector. La obra está respaldada por la experiencia del equipo de Abstracta y ofrece una perspectiva enriquecida y actualizada sobre las mejores prácticas en el área. Dirigido a profesionales del *testing*, este libro es una referencia esencial que no solo busca mejorar las habilidades técnicas de los *testers*, sino también potenciar su capacidad de comunicación y su comprensión integral de los sistemas de información. Es un libro que he recomendado ampliamente a todos los *testers* hispanos que he conocido por su gran cobertura, utilidad práctica, referencias e idioma.

Antonio Jiménez, especialista en Pruebas de *Performance*

Federico Toledo muestra en su libro cómo lograr eficiencia en la entrega de *software* de calidad. Su estilo de escritura es claro y accesible, utilizó un lenguaje amigable y ejemplos prácticos que hacen que los conceptos sean fáciles de entender, como si te los explicara un amigo.

Blanca Moreno, directora de QAminds

Si bien ya conocía a Fede por su presencia en eventos, en septiembre del año 2022 fue cuando lo pude conocer en persona. Desde esa primera charla en el patio del Konex en la conferencia Nerdearla, sentí una conexión especial, conversamos como si nos conociéramos de años, eso no se da con todo el mundo.

Recuerdo que unas personas se acercaron, lo reconocieron y agradecieron por su libro mencionando cuán importante fue para ellos. En ese momento, empecé a entender el impacto que genera y generó en la comunidad.

Mientras almorzábamos luego del Testing UY 2024, me mencionó este proyecto. Me pone feliz que sea una realidad y que 10 años después la comunidad siga creciendo acompañada de esta nueva edición.

Carlos Gauto, especialista en Automatización de Pruebas

Como responsable de Abstracta Academy y ReconverTite, he visto de primera mano el impacto profundo que este libro ha tenido en nuestra comunidad. Lo he recomendado durante todos estos años como una lectura fundamental, y puedo afirmar que ha sido un faro para quienes, sin conocimientos previos, querían aprender más sobre *testing*. Con este libro, Fede no solo ha inspirado a través de sus páginas, sino que también ha permitido la creación de diversos cursos que hoy forman parte de nuestro catálogo y de diversos proyectos de impacto social que llevamos adelante, ofreciendo a miles de estudiantes y profesionales la oportunidad de adquirir habilidades esenciales día a día. Agradezco profundamente su contribución, que ha enriquecido enormemente nuestra comunidad y ha impulsado

el crecimiento profesional de muchos. A quienes aún no han tenido la oportunidad de leerlo, los invito especialmente a descubrirlo.

Florencia Ripa, coordinadora de Abstracta Academy y *ReconverTite*

Este libro es un referente en la formación de *performance* en mi país. A modo de anécdota, recuerdo que mi entrenamiento estuvo más en la práctica que en la literatura y en el momento de prepararme para explicar mis pruebas desde la definición había un nivel de abstracción que parecía más fácil hacer que explicar. La mayoría de textos, si no que todos los que encontraba, estaban en inglés, y de un idioma al otro se pierden muchos conceptos. Entre tanto, uno de los compañeros con los que nos formábamos para presentar nuestros proyectos llegó con el PDF haciendo referencia de que todo lo que necesitábamos entender estaba ahí, una puerta en nuestro idioma que pasaba de lo abstracto a lo familiar y que daba sentido a la práctica, pues los usuarios y el mundo se estaban convirtiendo en un mundo que dependía de la inmediatez. Irónicamente, para medir esta aceleración, se respondía con precaución y menor velocidad. Federico respondía a una demanda de la que era difícil de hablar y que iba creciendo a pasos de gigante, la demanda de pruebas de *performance* se hacía cada vez más grande y la necesidad de formar especialistas y tener un plan de entrenamiento no respetaba el tiempo de capacitar. Este libro se convirtió en material de referencia y, así mismo, de crecimiento para todo aquel que quisiera desarrollar su carrera en pruebas de *performance*.

Jenifer Sánchez, especialista en Pruebas de *Performance*

El primer y por mucho tiempo único libro referente a QA en español que yo podía recomendar cuando alguien me preguntaba por alguna bibliografía en ese idioma. Sin pensarlo, lo recomiendo hasta el cansancio debido a lo completo que es dando una introducción general y muy digerible a todo el campo de QA. ¡Claro, también incluye pruebas de carga!

Leandro Melendez, especialista en Pruebas de *Performance*

El libro *Introducción a las pruebas de sistemas de información* ha sido un pilar fundamental en mi desarrollo como *tester* de *software*. Contar con la oportunidad de tenerlo como material de estudio no solo consolidó mis conocimientos técnicos, sino que también me brindó una comprensión profunda y estructurada sobre la importancia de las pruebas en el ciclo de vida del *software*. Gracias a su enfoque práctico y claro, logré perfeccionar mis habilidades y adquirir una visión más amplia de mi rol dentro del equipo de desarrollo. Sin duda, lo considero una lectura imprescindible para cualquier persona que esté comenzando en esta área, y siempre lo recomiendo como una herramienta clave para comprender el mundo de las pruebas de *software*.

Lisandra Armas, especialista en *Testing* de Accesibilidad

Junto a la Dra. Beatriz Pérez Lamancha, tuve la suerte de codirigir la tesis doctoral de Federico Toledo, en la Universidad de Castilla-La Mancha. Creo que durante los años en que elaboró *Madinga: a methodology for automation testing integrating functional and non-functional aspects* aprendimos mucho los tres: yo, desde luego, al sumergirme junto con él en aspectos de pruebas no funcionales y en la utilización de casos de prueba funcionales para derivar casos de prueba de rendimiento, estrés, etcétera. Por tanto, el texto que el lector tiene ante sus ojos procede de un auténtico experto en pruebas no solo a nivel técnico, sino también teórico. En este texto, Federico explica de manera clarísima los conceptos más importantes sobre *testing* y profundiza en sus aspectos más prácticos. Como profesor de Ingeniería de *Software* en la universidad, creo que este es y debe ser un texto de referencia para todas las personas que estudien esta materia.

Macario Polo Usaola, profesor en la Universidad de Castilla-La Mancha

Durante estos diez años, *Introducción a las pruebas de sistemas de información* ha logrado mucho más que simplemente enseñar sobre *testing*. Ha sido fundamental en el fortalecimiento de nuestra comunidad, porque aborda de manera integral y práctica aspectos clave del *testing*, permitiendo que muchas personas se adentren

con facilidad en el mundo de las pruebas de *software*. Fede, gracias por ofrecernos una guía tan completa donde adicionalmente abor das aspectos profundamente humanos que subrayan la importancia de la excelencia personal en esta profesión.

Mercedes Quintero, coCEO Regional de Abstracta

¿10 años ya? Es increíble que haya pasado tanto tiempo, que este libro haya sido parte de la formación de tantos profesionales y que, a pesar de los avances tecnológicos, las bases del *testing* se mantengan tan estables. Esto último hace que este libro siga siendo totalmente relevante. Recuerdo alguna conversación en la que Fede me contó que este libro surgió con la intención de ordenar los artículos de su blog para que fuera fácil de seguir para quienes recién empezaban. ¡Gran contribución para que no solo saquen provecho quienes buscaban algo específico, sino también para los nuevos que estaban llegando y no sabían por dónde arrancar! Por todo esto, creo que este libro no solo es una buena introducción al *testing*, sino también un testimonio del poder del conocimiento compartido.

Nadia Cavalleri, especialista en Pruebas y Calidad de *Software*

Fui alguien que entendió relativamente tarde que el *tester*, como el ser humano, no existe aislado; somos una comunidad. Para darme cuenta de esto, tuve que mudarme al otro lado del mundo, a Nueva Zelanda, donde comencé a sentir el deseo de compartir y leer en mi idioma sobre mi profesión. Fue allí donde encontré, en primer lugar, el blog de Fede. Me impactó especialmente un *post* sobre la necesidad de sustentabilidad entre *seniors* y *juniors* en la industria. De repente, se convirtió en un vicio leer lo que compartía; ise había convertido en mi referente en español número uno! No pasó mucho tiempo hasta que terminé leyendo su libro en PDF (hubiese amado tener una copia física), aquí, entre pohutukawas y playas. La claridad y familiaridad con la que comparte conocimientos y experiencias hacen que leer el libro sea una tarea muy sencilla, nada tediosa, como puede ocurrir con otros libros técnicos. Hoy en día, es el libro

número uno que recomiendo a mi comunidad para que comiencen sus viajes en *testing de software*.

Patricio Miner, especialista en Automatización de Pruebas

Precedente

Desde 2014, cuando compartimos la primera edición del libro, nunca imaginamos cuánta repercusión podría causar. Hoy, en 2024, si bien no están contabilizados, podemos afirmar que recibimos cientos de mensajes de cariño y agradecimiento. Nos los manifestaron tanto personas que comenzaron sus carreras como *testers* gracias al libro, como docentes que han creado y dictado cursos con el libro como guía.

De este modo, han amplificado su alcance y ayudado a cientos de estudiantes en sus inicios en el mundo del *software*. También sabemos que fue utilizado como material de referencia en muchas universidades, instituciones educativas, organizaciones no gubernamentales y más.

Todo esto, sin dudas, superó lo que proyectamos que podría aportar, y se traduce como nuestra mayor gratificación. Nos alegra enormemente que la primera edición haya resultado de tanta utilidad y esperamos que esta nueva versión siga por el mismo camino.

A quién está dirigido este libro

La actual publicación busca ayudar a aprender las bases del *testing* de forma amena. Está dirigida tanto a aquellas personas que aún no se familiarizaron con el tema como a quienes ya lo conocen e incluso se encuentran trabajando como *testers*, ya que presenta contenidos para profesionalizar y tratar con profundidad diversas temáticas.

El libro está organizado en capítulos sobre los temas que, según nuestra experiencia, son los más vistos y requeridos, al menos en los primeros años de trabajo en esta disciplina.

En las páginas que siguen, compartiremos parte del conocimiento que hemos ido adquiriendo en estos años de trabajo en la industria del *testing*, complementado con investigación sobre temas específicos que nos gustaría que puedan aprender.

Luego de leer este material, podrán entender qué es la calidad del *software* y cómo el *testing* puede colaborar en procura de ella, así como los principales desafíos que hay al respecto y posibles soluciones. También podrán acceder a un manejo sobre distintas técnicas de *testing*, dado que el libro incluye ejercicios y ejemplos prácticos para aplicar.

De este modo, podrán aprender sobre herramientas avanzadas que permiten mejorar las actividades de *testing* gracias a la automatización aplicada a distintos fines, como las pruebas de regresión automáticas y las pruebas de *performance*, que simulan el acceso de múltiples usuarios de manera simultánea.

El objetivo final de este libro es generar un impacto en la vida de quienes lo lean, ya sea para ofrecer conocimiento que les permita el acceso a oportunidades laborales en el ámbito tecnológico o para mejorar su desempeño en roles actuales, con el apoyo de un ejemplar de referencia para los temas que se necesita aplicar en el día a día.

Más allá de lo noble que pueda ser el fin, lo importante también es el camino, por lo cual anhelamos que todas las personas puedan disfrutar de su lectura mientras aprenden más sobre este apasionante tema del *testing* de *software*.

Sobre la tercera edición

Este es un libro con un carácter divulgativo. Queremos plasmar conceptos, buenas prácticas, técnicas y metodologías que hemos investigado, que nos han resultado útiles y creemos que les pueden servir a muchas más personas.

Está escrito en primera persona del plural porque, más que de un único autor, refleja el resultado de un equipo de trabajo. Nada de lo que se presenta aquí es

completamente propio y al mismo tiempo todo lo es, porque lo hicimos parte de nuestra cultura de *testing*. Lo adaptamos según nuestros criterios y necesidades.

Como se trata de un libro introductorio, no buscamos proporcionar un material sumamente extenso, sino que priorizamos los conceptos, técnicas y métodos que nos resultan de mayor relevancia (incluso en la redacción, estamos aplicando un enfoque no exhaustivo, no se intenta cubrir todos los temas, sino que se priorizan solo los más importantes).

Quienes nos conocen, han escuchado charlas nuestras o leen algunos de los blogs en los que participamos (por ejemplo, el blog de Abstracta en español¹, mi sitio *web* en español² o el blog de Abstracta en inglés³), seguramente se darán cuenta de que muchas de estas cosas ya las hemos nombrado en otras ocasiones. Lo que estamos haciendo aquí es organizar el contenido y escribirlo con más formalidad (bueno, quizá no tanta formalidad, veremos).

La primera edición fue publicada en 2014: imprimimos 500 copias que se terminaron de regalar o vender a precio de costo de impresión en 2018. En 2015, imprimimos una segunda edición en México, con cambios menores, como parte de un proyecto que hoy es Abstracta Academy⁴. Esta **tercera edición** cuenta con una revisión y varios ajustes, que surgen de algunas nuevas visiones que hemos adoptado en estos años, así como de revisiones del nuestro lenguaje, que tratamos de mejorar día a día.

Hizo falta realizar algunos cambios en ejemplos basados en herramientas que quedaron un poco desactualizadas. Asimismo, hemos realizado ajustes en base al *feedback* que recibimos en las primeras ediciones, lo cual se ha constituido como el eslabón más valioso de todo este proceso. Por ejemplo, todos los *links* que aparecen en el libro se pueden encontrar en este *post*: federico-toledo.com/libro.

¹ Blog de Abstracta en español: <<https://es.abstracta.us/blog>>

² Sitio *web* personal de Federico Toledo: <www.federico-toledo.com>

³ Blog de Abstracta en inglés: <www.abstracta.us/blog>

⁴ Sitio *web* de Abstracta Academy: <<https://abstracta.academy>>

Agradecimientos

Agradecemos a todas las personas que nos han hecho crecer, tanto a quienes integran o han integrado alguna vez el **equipo de Abstracta**, como a amistades y **colegas** de empresas *partners* y clientes. A todos y todas, ¡muchas gracias!

En particular, quisiera agradecer a Andrés Curcio y Giulliana Scuoteguazza, quienes me ayudaron a escribir un capítulo cada uno en la primera edición. Sin su contribución, el libro no contaría con tanta variedad de temas y puntos de vista. Para esta tercera edición en particular, varias personas han aportado en diferentes revisiones y sugerencias. Agradezco principalmente a Alejandro Berardinelli, Andrei Guchín, Martín Acuña, Natalie Rodgers, Sebastián Lorenzo y Verónica Gamarra por sus aportes en las revisiones de esta edición.

Por último, pero no por ello menos importante, un reconocimiento **a nuestras familias**, a quienes nos “hacen el aguante” a diario en esta cruzada de evangelizar con el *testing* y la calidad, ya sea con Abstracta, universidades, *meetups* y eventos, etc.

CONTENIDO

PRÓLOGO	3
INTRODUCCIÓN	15
¿Es posible construir un <i>software</i> que no falle?	15
La calidad, el <i>testing</i> y sus objetivos	17
Tipos de pruebas	20
El <i>testing</i> al final del proyecto	23
Falacias sobre el <i>testing</i>	25
<i>Testing</i> independiente	27
Métricas de <i>testing</i>	29
Estructura del resto del libro	31
INTRODUCCIÓN A LAS PRUEBAS FUNCIONALES	33
¿Qué tareas ocupan a las personas que hacen pruebas?	34
Conceptos introductorios de <i>testing</i>	40
Diseño básico de casos de prueba	47
Comentarios finales del capítulo	64
TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA	65
Derivación de casos de prueba desde casos de uso	66
Técnica de Tablas de Decisión	79
Técnicas de Grafos Causa-Efecto	84
Técnica de Máquinas de Estado	89
Técnica de Matriz CRUD	97
Comentarios finales del capítulo	100
INTRODUCCIÓN AL TESTING EXPLORATORIO	101
¿Qué es el <i>Testing</i> Exploratorio?	102
Trabajando con <i>Testing</i> Exploratorio Basado en Sesiones	109
Comentarios finales del capítulo	120

AUTOMATIZACIÓN DE PRUEBAS FUNCIONALES	121
Introducción a las pruebas automatizadas	122
Principios básicos de la automatización de pruebas	134
Buenas prácticas de automatización de pruebas	151
BDD (<i>Behavior-Driven Development</i>)	162
Ejecución de pruebas automáticas	166
Estrategia de automatización	170
Comentarios finales del capítulo	172
 PRUEBAS DE <i>PERFORMANCE</i>	 175
Introducción a las pruebas de <i>performance</i>	176
Diseño de pruebas de <i>performance</i>	182
Preparación de pruebas de <i>performance</i>	195
Ejecución de pruebas	200
Falacias del Testing de Performance	214
Comentarios finales del capítulo	222
 HABILIDADES PARA TESTING	 225
¿Qué habilidades necesitamos desarrollar para testing?	226
Habilidades personales vs. técnicas	236
Pasión y motivación	239
Comentarios finales del capítulo	240
 CIERRE	 241

INTRODUCCIÓN

¿Es posible construir un *software* que no falle?

Con motivo del Año de Turing, celebrado en 2012, el diario *El País* de España publicó un artículo muy interesante, en el cual cuestionó si es posible construir sistemas de *software* que no fallen. Este tema cobra especial relevancia en la actualidad, en un mundo casi completamente interconectado, en el cual todos los ámbitos de la vida están mediados por la tecnología. Actualmente, las personas estamos acostumbradas a la mala calidad del *software*, a que eventualmente todos los sistemas de los cuales dependemos fallen. El problema se presenta cuando estos errores producen grandes tragedias, muertes, accidentes, pérdidas de miles de dólares, etc. Desafortunadamente, existen miles de ejemplos de ello.

El artículo comienza con ejemplos de catástrofes (típico en todo profeta del *testing*) y luego plantea un caso real que resulta esperanzador respecto a que exista un *software* “perfecto”, o al menos suficientemente bueno como para no presentar fallos que afecten a los usuarios. Este sistema que no ha presentado fallos durante años ha sido desarrollado con un lenguaje basado en especificación de reglas de negocio, con las que se genera el código del sistema y de su verificación automática.

El ejemplo planteado es el del sistema informático de la línea 14 del metro de París⁵. Fue la primera en estar completamente automatizada. ¡Los trenes no tienen conductor! Son guiados por un *software* y mucha gente viaja por día (al final del

⁵ Información sobre el metro de París número 14. (2024):
<http://es.wikipedia.org/wiki/L%C3%ADnea_14_del_Metro_de_Par%C3%ADs>

2007, un promedio de 450.000 pasajeros tomaba este medio en un día laboral). Actualmente se ha expandido a más líneas.

Según cuenta *El País*, el sistema tiene 86.000 líneas de código ADA⁶ (definitivamente no es ningún *hello world*⁷). Fue puesto en funcionamiento en 1998 y hasta 2007 no presentó ningún fallo.

¿A qué queremos llegar con esto? A que es posible pensar que somos capaces de desarrollar *software* suficientemente bueno. Es probable que haya errores en este sistema, pero no se han manifestado o no son tan importantes como para que los usuarios o el negocio se vean afectados. Eso es lo importante. ¿Y cómo ayudamos a que se genere *software* de calidad suficiente? Obviamente, lo que vamos a decir es que con un *testing* suficientemente bueno. Entonces, debemos ser capaces de verificar los comportamientos del sistema que son más típicos, que pueden afectar más a quienes lo usen o que son más importantes para el negocio. Esto estará limitado o potenciado por otros factores más relacionados al mercado, pero nos centraremos en cómo “asegurar la calidad”⁸, y optimizar los costos lo mejor posible; dejaremos esas otras discusiones al margen.

Esta visión es un poco más feliz que la que siempre escuchamos de Dijkstra⁹, que indica que el *testing* nos sirve para mostrar la presencia de fallos, pero no la ausencia de ellos. ¡Lo cual es muy cierto! Pero no por esta verdad vamos a restarle valor. Lo importante es hacerlo en una forma que aporte valor y nos permita acercarnos a ese *software* perfecto o, mejor dicho, de calidad.

⁶ Lenguaje de programación ADA:

<[https://es.wikipedia.org/wiki/Ada_\(lenguaje_de_programaci%C3%B3n\)](https://es.wikipedia.org/wiki/Ada_(lenguaje_de_programaci%C3%B3n))>

⁷ Con *hello world*, nos referimos al primer programa básico que generalmente se implementa cuando alguien experimenta con una nueva tecnología o lenguaje de programación.

⁸ *Asegurar la calidad* se escribió entre comillas a propósito. ¿Alguien puede asegurar la calidad?

Hay un artículo de Michael Bolton (2010) muy interesante al respecto, en el que indica que quizá el que pueda hacer eso es quien ocupe el rol de CEO de la compañía, responsable de manejar todos los recursos y asignaciones, tomar las grandes decisiones. Como *testers*, lo máximo que podemos hacer es asegurar la información oportuna para que esas decisiones se tomen en mejores condiciones.

Estamos de acuerdo con su visión:

<<http://www.developsense.com/blog/2010/05/testers-get-out-of-the-quality-assurance-business>>

⁹ Información sobre Edsger Dijkstra: <https://es.wikipedia.org/wiki/Edsger_Dijkstra>

La calidad, el *testing* y sus objetivos

El objetivo del *testing* es brindar información que permita tomar mejores decisiones para construir un producto de calidad. Todo en una frase: *calidad, testing y objetivo*. Veamos un poco a qué se refiere cada término para entrar en detalles.

Calidad

Una pregunta que resulta muy interesante es la que cuestiona qué es *la calidad*. Existe una ley divina que indica que todos los cursos, seminarios, tutoriales o materias electivas de carreras universitarias de Ingeniería relacionadas con *testing* deben iniciarse con ese cuestionamiento. Así es que, cada vez que nos realizan esta pregunta, debemos detenernos a pensar nuevamente en ello, como si fuese la primera vez que analizáramos el tema.

Siempre se vuelve difícil llegar a un consenso, pero hay un punto en el que todas las personas solemos estar de acuerdo: se trata de un concepto muy **subjetivo**.

Podríamos decir que *la calidad* es una **característica que nos permite comparar distintas cosas del mismo tipo**. Se define, se calcula o se le asigna un valor en base a un conjunto de propiedades (seguridad, *performance*, usabilidad, etc.) que podrán ser ponderadas de distinta forma. Lo más complicado del asunto es que para cada persona y para cada situación, seguramente, la importancia de cada propiedad será distinta. De esta forma, algo que para nosotros es de calidad, tal vez para otras personas no lo sea. Algo que hoy consideramos de calidad, quizá el año próximo ya no lo sea, incluso para nosotros mismos. Complejo, ¿cierto?

Según Jerry Weinberg, “la calidad es valor para una persona”. Esto fue extendido por Cem Kaner de la siguiente manera: “la calidad es valor para una persona a la que le interesa”.

En relación a este tema, existe una falacia conocida como *Falacia del Nirvana* que se refiere al error lógico de querer comparar cosas reales con otras irreales o idealizadas¹⁰. Un punto muy negativo sobre esto es la tendencia de pensar que las opciones reales son malas por no ser perfectas, como las idealizadas. Debemos asumir que **el software no es perfecto** y no compararlo con una solución ideal inexistente, sino con una solución que dé un valor real a las personas que lo utilizarán. También podemos hacer referencia a la famosa frase de Voltaire, que dice que “lo mejor es enemigo de lo bueno”, la cual también hemos escuchado como “lo perfecto es enemigo de lo bueno”.

Testing

Según Cem Kaner, el *testing* consiste en una investigación técnica realizada para proveer información a los *stakeholders* sobre la calidad del producto o servicio bajo pruebas. Según Glenford Myers, se trata de un proceso diseñado para asegurar que el código haga lo que se supone que debe hacer, y no haga lo que se supone que no debe. Según este autor, es un proceso en el que se ejecuta un programa con el objetivo de encontrar errores. Lisa Crispin y Janet Gregory lo plantean de esta forma: el *testing* en un equipo ágil es una actividad colaborativa que consiste en trabajar juntos para entregar el *software* correcto, con la calidad correcta, de forma más rápida. Otra forma de conceptualizar que nos gusta es la de Maaret Pyhäjärvi, ella siempre hace referencia a que el *testing* es una actividad de investigación en la que nos enfocamos en aprender sobre el producto a través de la interacción y la experimentación.

Lo más importante: aportar al proceso de generar un producto de calidad. ¿Cómo? Detectando posibles incidencias de diversa índole que se puedan presentar cuando

¹⁰ Información sobre Falacia del Nirvana: <http://es.wikipedia.org/wiki/Falacia_del_Nirvana>

esté en uso y riesgos, haciendo preguntas (incómodas muchas veces), y empatizando con los posibles usuarios.

Objetivos

La forma en la que estaremos “aportando calidad” será principalmente buscando fallos¹¹. Por supuesto. Digamos que, si no encontramos fallos, todo el costo del *testing* nos lo podríamos haber ahorrado. ¿No? ¡No! Porque si no encontramos fallos, también estamos aportando información relacionada a la calidad. Entonces, el equipo sentirá más confianza en que los usuarios encontrarán menos problemas.

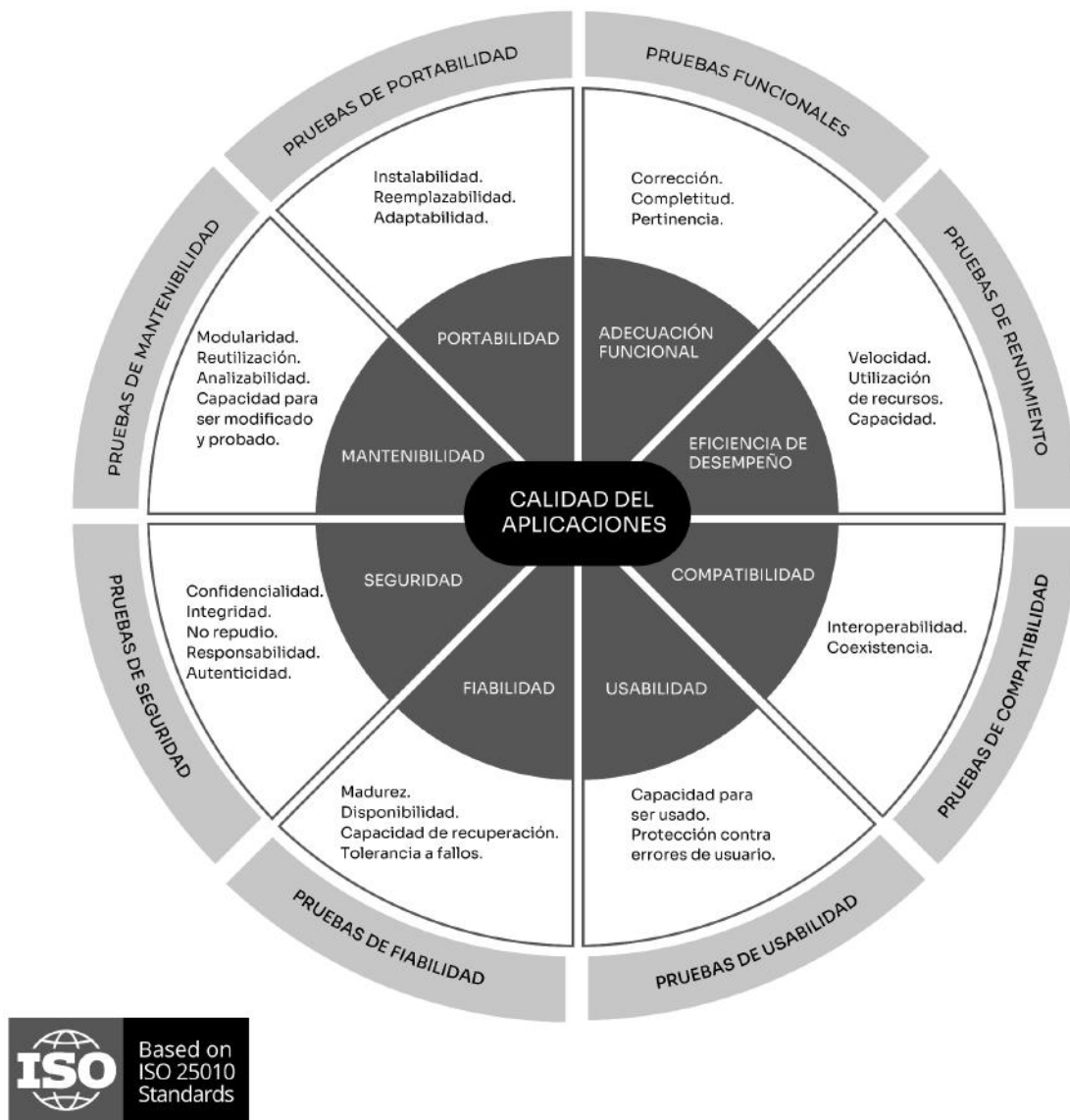
¿Se trata de encontrar la mayor cantidad de fallos posible? ¿La persona que encuentra 100 fallos en un día hizo mejor su trabajo que una que apenas encontró 15? Definitivamente, no. Si esto fuera así, la estrategia más efectiva sería centrarse en el módulo que esté “más verde” (menos maduro) y que tenga errores más fáciles de encontrar, pues es justamente ahí en donde se puede hallar una mayor cantidad de fallos. Pero, **¡no! No es de ese modo**, sino que la idea es encontrar la mayor cantidad de fallos que más calidad le aporten al producto al ser corregidos. En otras palabras: los que, al ser usados, más van a molestar (entorpecer, agobiar, enloquecer, enfurecer, etc.), todos esos sentimientos negativos que a veces nos generan las aplicaciones y que nos hacen fruncir el ceño).

¡Ojo! El objetivo no tiene por qué ser el mismo a lo largo del tiempo, pero sí es importante dejar claro cuál es en cada momento. Puede ser válido en cierto momento definir como objetivo encontrar la mayor cantidad de errores posibles, entonces, seguramente el *testing* se enfoque en las áreas más complejas del *software* o “más verdes”. Si el objetivo es dar seguridad a quienes usen la herramienta, entonces probablemente se enfoque en los escenarios más requeridos por clientes.

¹¹ Cuando hablamos de *fallos* o *bugs* nos referimos en un sentido muy amplio que puede incluir desde errores, diferencias con la especificación, oportunidades de mejora desde distintos puntos de vista de la calidad: funcional, *performance*, usabilidad, estética, seguridad, etc.

Tipos de pruebas

Al hablar de calidad, en la sección anterior decíamos que hay una lista de propiedades por evaluar en las aplicaciones de *software*. Ahora, ¿cuáles son esas propiedades? Veamos el siguiente modelo que diseñamos, al cual le llamamos *la Rueda del testing*.



Existe una norma ISO que plantea cuáles son los factores de calidad de un producto de *software*, la norma ISO 25000¹². En base a eso, diseñamos el modelo anterior, para mostrar cada uno de esos factores, y cómo para cada factor de calidad existe un tipo de prueba. Es así como, para seguridad, existe el *testing* de seguridad; para usabilidad, existen las pruebas de usabilidad, etc. En cierta forma, se refleja la idea de que las actividades de *testing* dejan protegida cada una de las características de calidad que hay en el centro.

Lo importante es que para cada factor de calidad podemos hacer experimentos que nos permitan evaluar estas propiedades, enfocarnos en encontrar información y detectar riesgos relacionados con cada característica.

Se pueden distinguir entre estos atributos algunos que son de calidad interna y otros de calidad externa. Por ejemplo, la usabilidad sería externa, porque es algo que una persona puede percibir fácilmente. Por otro lado, la mantenibilidad sería algo interno, ya que no se puede percibir directamente, está relacionada con la calidad del código, lo cual no está al alcance de la mano de quien use la herramienta. De todos modos, una salvedad que es importante destacar: si algo tiene poca mantenibilidad, llega un momento en que ese factor interno se vuelve externo. Es decir, la mala mantenibilidad se hace perceptible. Por ejemplo, cuando alguien pide un cambio simple en el sistema y se tarda meses para resolverlo, ese factor interno le termina impactando directamente.

De aquí que se vuelve fundamental prestar atención a todos los factores, pero más que nada saber cuáles son más importantes para el negocio en cada momento y cuáles no tanto.

Otra consideración interesante es que los distintos factores están interrelacionados. Hay una historia que refleja esto y nos gusta contarla, al margen de no saber si es verídica o no. Se trata de un ascensor que fue motivo de quejas por su lentitud. La solución que le dieron al problema no fue cambiar el motor por uno más potente, sino que, simplemente, agregaron espejos en su interior. Esto hizo que las personas que lo usaban dejaran de quejarse. Claramente, esta solución no mejoró la velocidad, pero sí la percepción sobre cuál era la calidad, ya que dejaron de darle

¹² Norma ISO 25000: <<https://iso25000.com>>

tanta importancia a la velocidad, porque podían aprovechar el tiempo dentro del ascensor para revisar su apariencia y acomodarse la ropa o el peinado. Por un lado, esto es un nuevo reflejo de que la calidad no es absoluta, sino que es medida según los ojos del usuario, y por otro lado muestra que distintos factores (velocidad, funcionalidades, experiencia de usuario) están interrelacionados. También es interesante pensar que no podemos trabajar cada factor de calidad aisladamente, sino que tenemos que tener un enfoque holístico de la calidad.

Esta interrelación muchas veces nos obliga a tomar decisiones en base a cuál de los factores de calidad es más prioritario para nuestro negocio, nuestro producto y, más que nada, nuestros usuarios. Por ejemplo, hay ocasiones en las que mejorar la seguridad puede empeorar la usabilidad, como es el caso de agregarle más requerimientos a la clave del *login*, pidiendo que se incluya una mayúscula, una minúscula, más de ocho caracteres, etc. Mientras más requisitos se agreguen, se estará mejorando la seguridad, pero al mismo tiempo se provocará que la facilidad de uso o incluso la accesibilidad se comprometan.

El *testing* al final del proyecto

Scott Barber solía decir en sus charlas algo específico para pruebas de *performance*, pero que se puede extrapolar al *testing* en general: “Hacer las pruebas al final de un proyecto es como pedir examen de sangre cuando el paciente ya está muerto”.

La calidad no es algo que se agrega al final como quien le pone azúcar al café. En cada etapa del proceso de desarrollo habrá actividades de *testing* para realizar que aportarán en distintos aspectos de la calidad del producto. Si las dejamos para el final, nos insumirá mucho más: más tiempo, más horas y más costo de reparación. Por ejemplo, si encontramos un problema en la arquitectura del sistema cuando este ya se implementó, entonces, los cambios serán más costosos.

Ahora, ¿por qué no lo hacemos así siempre? Acá hay un problema de raíz. Ya desde nuestra formación en carreras de Ingeniería aprendemos un proceso un poco estricto, que se refleja bastante en la industria¹³:

- Primero, matemáticas y materias formativas que nos ayudan a crear el pensamiento abstracto y lógico.
- Programación básica, estructurada y usando constructores fundamentales.
- Programación con tipos abstractos de datos.
- Programación con algoritmos sobre estructuras de datos.
- Programación orientada a objetos (por ejemplo, en C++).
- Un lenguaje con más abstracción (por ejemplo, Java).
- Sistemas operativos, arquitectura y otras de la misma rama.
- Luego un proceso pesado, como lo es el RUP¹⁴, poniéndolo en práctica en un grupo de 12 a 14 personas. Recién en estas materias se ve algo de *testing*.
- Para los últimos años suele haber electivas específicas de *testing*.

¹³ Basado principalmente en nuestra experiencia en la Universidad de la República en Uruguay, pero ya hemos conversado con distintos colegas en otros países —como en Argentina y España— y comprobamos que la realidad no se diferencia mucho de este esquema. Por suerte, ha estado mejorando en los últimos años.

¹⁴ Información sobre RUP: <<http://es.wikipedia.org/wiki/RUP>>

¿Qué nos deja esto? Que el *testing* es algo que queda para el final y, además, opcional. ¡Oh, vaya casualidad! ¿No es esto lo que pasa generalmente en el desarrollo? Por eso, se habla tanto de *la etapa de testing*, cuando en realidad no debería ser una etapa, sino que debería ir de la mano con el desarrollo en sí. El *testing* debería ser un conjunto de distintas actividades que acompañen y retroalimenten al desarrollo desde un principio.

En los últimos años, relacionado a las metodologías ágiles, DevOps, *Continuous Integration* y *Continuous Delivery*, se ha estado hablando mucho de *shift left testing*, *shift right testing* y *continuous testing*. O sea, “mover” el *testing* a etapas más tempranas del desarrollo (a la izquierda), no dejar de probar luego de enviar a producción, “moviendo” el *testing* a etapas más tardías (a la derecha) y así generando una cultura de *testing* continuo. Lo importante también es que no hay que ver al *testing* como una actividad pura y exclusiva de *testers*. Hay muchas actividades de *testing* que deberían ser realizadas por otros integrantes del equipo y, en particular, por personas dedicadas al desarrollo. Todo el equipo es responsable de la calidad (no existe “un” responsable de la calidad, ni un equipo responsable de la calidad, que solía ser el equipo de *testers*). Cada quien, desde su rol, incide para que el resultado íntegro sea un producto de buena calidad.

Vale destacar que es un proceso empírico, se basa en la experimentación, mediante la que se le brinda información sobre la calidad de un producto o servicio a alguien interesado al respecto.

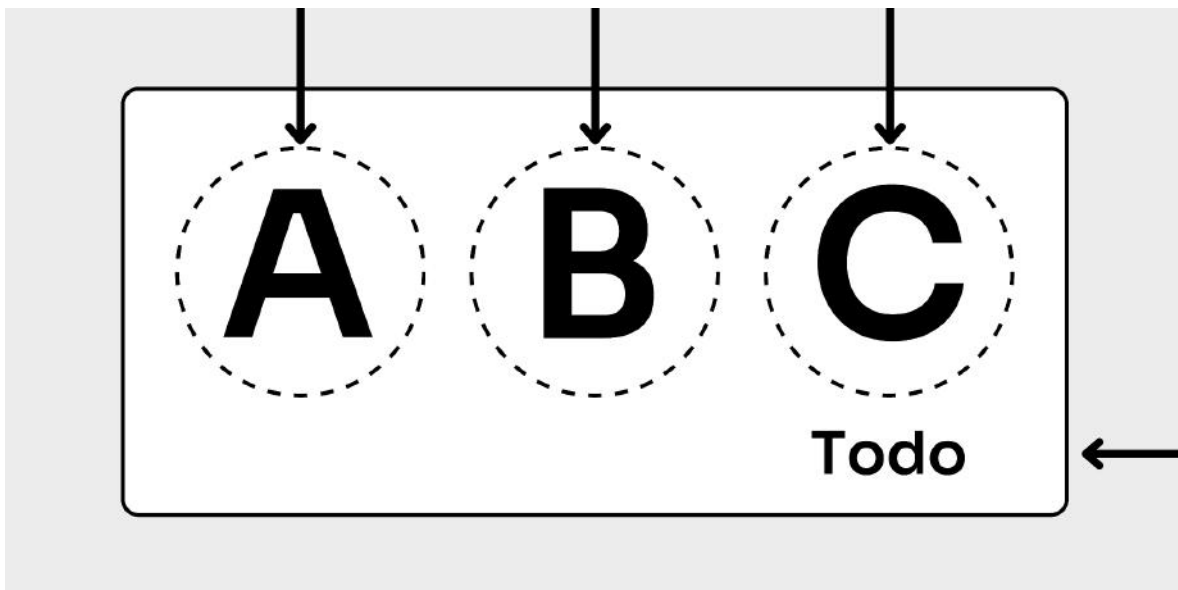
Falacias sobre el *testing*

Existen algunas falacias que habitualmente se relacionan y resuenan bastante en el *testing* de *software*. Las que nos interesa mencionar son:

- La falacia de la descomposición.
- La falacia de la composición.
- La falacia de que “todo *testing* es... *testing*”.

Primero, según el *Diccionario de la lengua española* (2014), una *falacia* se define como “un engaño, mentira o falsedad con la que se intenta engañar a alguien”. También se puede encontrar definiciones que dicen que “es un razonamiento no válido o incorrecto que, con apariencia de exactitud, pretende ser persuasivo”.

Uno de los libros relacionados al *testing* que nos gustaría recomendarles es *Perfect software: And other illusions about testing*, de Jerry Weinberg (2008). En uno de sus capítulos denomina a varias experiencias y falacias que en más de una ocasión hemos vivido y volveremos a vivir profesionalmente.



Falacia de la composición: $\text{test}(A) + \text{test}(B) + \text{test}(C) = \text{test}(\text{Todo})$

Falacia de la descomposición: $\text{test}(\text{Todo}) = \text{test}(A) + \text{test}(B) + \text{test}(C)$

En la figura se representa la idea de la falacia de la descomposición y de la composición, por la que se tiende a creer que, si a un sistema lo probamos como un todo, entonces, también podemos considerar que probamos sus partes o que, si probamos las partes, entonces, podemos dar el sistema en su integridad como probado. La realidad es que el hecho de que probemos un sistema como un todo no asegura que probemos el correcto funcionamiento de las partes individuales, y viceversa. Si solo probamos las unidades de acuerdo a su especificación, no sabremos cómo estas van a interactuar con el resto de unidades y, por esto, se terminará generando problemas en el sistema integrado. Si solo probamos el sistema completo, ¿cuánto podemos decir que cubrimos de las unidades que la componen? Solo estaremos probando las unidades en determinado contexto de uso y, quizá, al ser utilizadas en otros contextos puedan presentar diversos problemas que no se hicieron evidentes en nuestras pruebas del sistema integrado.

Por último, una de las falacias más conocidas: *todo testing es... testing*. Es decir, apoyar la idea de que toda y cualquier *acción*, entiéndase “presionar cualquier tecla del teclado o clic del *mouse*”, es *testing*. Para peor, sin importar si estamos probando los requerimientos, la intención por la que se establecieron, la implementación de las unidades o la integración con otros sistemas.

Testing independiente¹⁵

Imaginen un proyecto cualquiera en el que un equipo de desarrollo está construyendo un *software*. Todo comienza muy bien, pero las referidas personas, al momento de realizar pruebas por sí mismas, hacen evidente que existe un conflicto de intereses inherente a cualquier proyecto. Se suele pedir a la gente que construyó el *software* que lo pruebe. No parece haber nada raro en ello, pues quienes construyeron el *software* son los que mejor lo conocen. El problema es que, así como cualquier profesional que siente orgullo por su obra intenta evitar que le encuentren defectos, las personas que programan están interesadas en demostrar que el programa está libre de errores y que funciona de acuerdo con las especificaciones de clientes, habiendo sido construido en los plazos adecuados y con el presupuesto previsto.

Tal como lo ha dicho alguna vez Antoine de Saint-Exupery (1943): “Es mucho más difícil juzgarse a sí mismo que juzgar a los demás”.

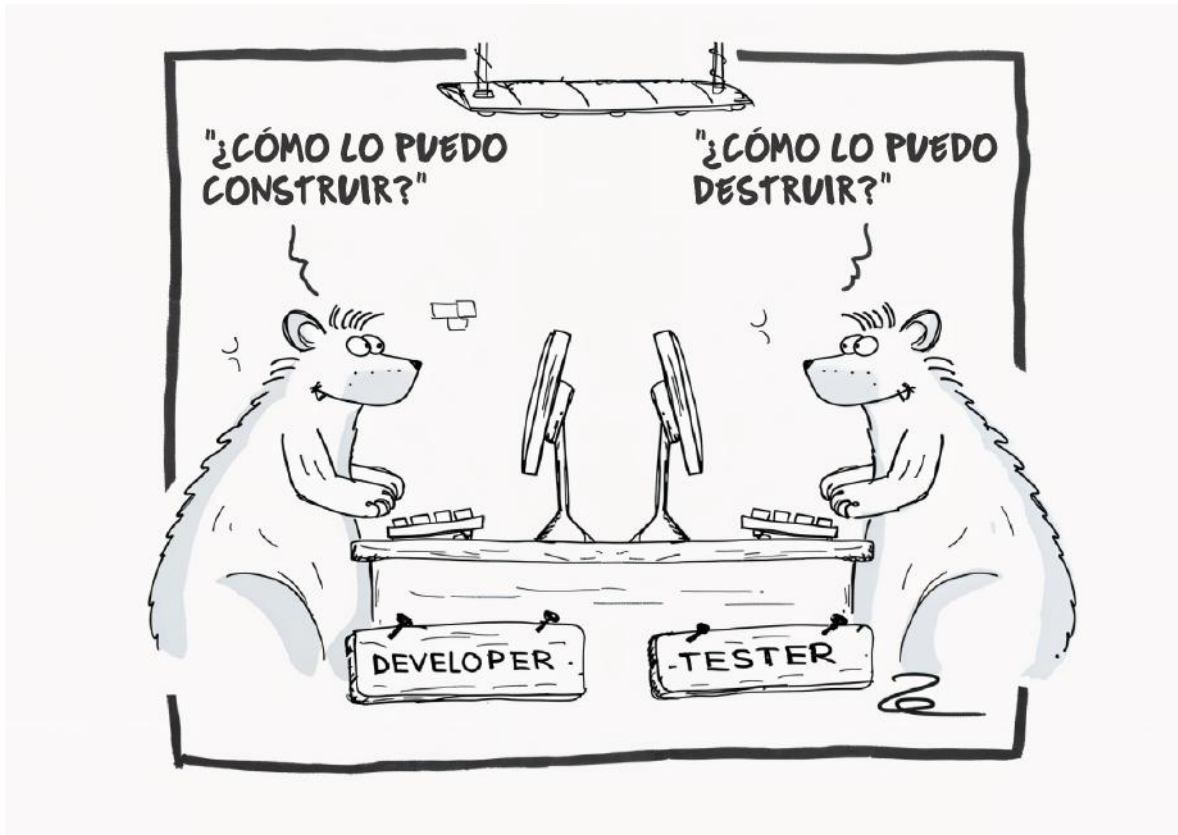
Desde el punto de vista de quien programa (responsable de construir el *software*), quien prueba se rige por un enfoque destructivo, pues intenta destruirlo (o al menos mostrar que no funciona como es debido). Si bien es cierto que el enfoque de la persona que prueba debe ser destructivo, también es cierto que sus aportes igualmente forman parte de la construcción del *software*, ya que el objetivo final es aportar a su calidad.

También hay otro tema de percepción, arrogancia o como se quiera llamar, que tiene que ver con que se concibe el rol de *testing* para una persona con menos conocimientos o experiencia, para quien no sabe programar, que no participa en la construcción, por lo que se asume que no entiende. Imaginemos cómo se debe sentir alguien que desarrolló un sistema cuando viene esa persona que supuestamente sabe menos (*tester*) a marcar errores...

La siguiente caricatura representa la idea de que somos las mismas ratas de laboratorio, pero tal vez estamos enfrentados en cuanto a la forma de ver o de

¹⁵ Basado en Roger Pressman (2010).

pensar sobre el mismo problema. Mientras quien trabaja en desarrollo se concentra en “cómo lo puedo construir”, la persona dedicada a *testing* piensa en “cómo lo puedo destruir”.



Pero atención: *testing* (como equipo, persona o personas) es un aliado, un amigo y un cómplice en el armado del producto. Entonces, a no malinterpretar lo de la visión destructiva, porque en realidad su aporte es completamente constructivo!

Métricas de *testing*

En el mundo del *testing*, las métricas juegan un papel crucial al proporcionar información valiosa que ayuda a reducir incertidumbres y controlar riesgos. Sin embargo, es esencial abordar con cuidado el uso de métricas, ya que pueden influir en el comportamiento y los resultados de manera inesperada. Tendremos métricas que nos ayudarán a transmitir información sobre el producto que estamos probando, sobre el *testing* que estamos haciendo y así dar visibilidad del progreso del trabajo, y muchas más.

Las métricas pueden desencadenar comportamientos no deseados, influyendo en la forma en que las personas actúan. Este fenómeno, conocido como *el efecto de la medición*, puede distorsionar los resultados. Por ejemplo, si se mide la cantidad de problemas reportados por un *tester*, es probable que este intente aumentar la cantidad de *bugs* reportados para mejorar así la métrica, lo que puede desviar la atención hacia áreas menos críticas, pero en las que es más fácil probar o encontrar errores.

Otro desafío radica en la facilidad con la que se pueden manipular las métricas y seleccionar indicadores que muestren lo que uno quiere mostrar.

Es fundamental reflexionar sobre qué métricas son realmente importantes para el negocio y enfocarse en mejorar aquellas que contribuyen directamente a los objetivos que persigue el equipo, la empresa, el negocio. Además, es crucial fomentar un compromiso generalizado del equipo hacia la calidad, en lugar de simplemente perseguir métricas aisladas.

Además de identificar qué métricas queremos incluir en nuestros reportes, será importante ver cómo vamos a obtener la información requerida para calcular los indicadores. Debemos también definir umbrales claros de los valores esperados para cada métrica y establecer planes de acción en caso de que los valores estén por fuera de los umbrales. Esto permitirá tomar decisiones mejores y orientadas a resultados.

Las métricas son herramientas valiosas para mejorar la visibilidad y la toma de decisiones, siempre y cuando se utilicen de manera estratégica y se alineen con los

objetivos del negocio. Es esencial recordar que las métricas deben tener un propósito claro y no medir por medir.

Estructura del resto del libro

Hasta acá vimos una introducción a temas fundamentales relacionados a calidad, *testing* y algunos aspectos clásicos sobre lo que es el foco principal del libro que nos permitirán entender mejor lo que está por venir.

En las próximas páginas, abordamos siete grandes temas o secciones relacionados a las pruebas:

- **Introducción a las pruebas funcionales:** comenzaremos por conceptos básicos e introductorios para seguir luego mostrando su utilidad, profundizando en un posible método básico para abordar la tarea de diseñar pruebas. Entre otras cosas, veremos las técnicas que incluso se terminan utilizando casi en forma inconsciente, tales como: partición en clases de equivalencia, valores límites, etc.
- **Técnicas de diseño de pruebas funcionales:** luego de haber visto las básicas de las pruebas funcionales, podremos seguir describiendo algunas técnicas a las que llamaremos *avanzadas*, siempre pensando en particular en sistemas de información. Entre ellas, veremos cómo derivar casos de prueba a partir de casos de uso, tablas de decisión, máquinas de estado, etc.
- **Pruebas exploratorias:** generalmente se las confunde con pruebas *ad-hoc*¹⁶, pero no se trata de no pensar o no diseñar pruebas, o no aplicar conocimientos en *testing*. Al contrario, es un enfoque bastante avanzado que consiste en diseñar y ejecutar al mismo tiempo. Veremos que existen escenarios especiales en los que nos resultará principalmente útil y también cómo podríamos aplicar la técnica en forma práctica.
- **Pruebas automatizadas:** veremos cómo la automatización aplicada en el *testing* nos puede aportar beneficios para bajar costos y aumentar la cantidad de pruebas que logramos ejecutar. Luego veremos qué tener en cuenta para no fracasar en el intento.
- **Pruebas de *performance*:** en este caso, trataremos el tema de cómo automatizar pruebas con el fin de simular múltiples usuarios concurrentes y

¹⁶ Información sobre *Ad-hoc*: <https://es.wikipedia.org/wiki/Ad_hoc>

así poder analizar el rendimiento de la aplicación bajo pruebas, sus tiempos de respuesta y el consumo de recursos.

- **Habilidades para *testing*:** porque no todo en la vida son solo técnicas y tecnologías, también queremos tocar algunos puntos más relacionados a los aspectos humanos que también debemos preocuparnos por desarrollar, especialmente relacionados a la capacidad de comunicar.

De esta forma, estamos abordando distintos aspectos que, desde nuestro punto de vista, son primordiales para el éxito de la implantación de un sistema de información y el desarrollo profesional de todo y toda *tester*.

El *testing* no es una varita mágica que arregla el *software* que se está probando.

Mónica Wodzislawski

INTRODUCCIÓN A LAS PRUEBAS FUNCIONALES

En esta sección analizaremos algunas de las tantas técnicas de diseño de casos de prueba y datos de prueba. En particular, el foco estará en las que especialmente aplican al probar sistemas de información con bases de datos. Luego de comenzar con algunos conceptos introductorios, veremos lo que podría ser una técnica o metodología básica y útil para la mayoría de los casos. Esto servirá principalmente a quienes comienzan a meterse en el mundo de las pruebas a seguir un método simple y efectivo; y a medida que adquieran experiencia podrán ir aprendiendo más técnicas y mejorando sus habilidades con absoluta confianza.

¿Qué tareas ocupan a las personas que hacen pruebas?

Antes de comenzar con definiciones, técnicas y todo eso, nos gustaría describir a grandes rasgos qué es lo que hace la persona con el rol de *tester*. ¿Qué implica hacer *testing*, hacer pruebas? ¿Cómo es un día en la vida de quien hace *testing*? Por supuesto que lo que van a leer a continuación es una primera aproximación, y según la empresa o equipo donde trabajen habrá más o menos actividades, pero al menos servirá de referencia.

Pruebas de una funcionalidad o cambio

En el día a día de un equipo de desarrollo, ya sea de un sistema nuevo o uno en mantenimiento (o sea, que esté en producción y se le vayan haciendo ajustes, cambios, mantenimiento), el equipo recibe *solicitudes*, *requerimientos*, *requisitos*, *órdenes de cambio* (esos quizá son los nombres más comunes). De alguna forma, el equipo debe organizar cómo implementará los cambios referidos. Estas formas de organizarse varían según la metodología que se utilice, como por ejemplo, si están trabajando en un esquema tradicional en cascada, o si usan una metodología ágil como Scrum, XP o Kanban (los términos que no conozcan y les generen curiosidad, ya saben, ¡a *googlearlos*!).

Según el equipo y sus formas de trabajo, la responsabilidad y el involucramiento de *testers* podrá variar, pero intentaremos acá presentarlo de forma genérica. En algún momento, conocerán qué es lo que se le plantea al equipo de desarrollo que se desea implementar. Esto generalmente se documenta, ya sea en una especificación formal de requerimientos, o en una breve descripción en una historia de usuario. De una forma u otra, quien va a desarrollar utilizará esa información para decidir cómo implementarla, y a su vez, quien va a probar utilizará esa información para pensar sus pruebas.

Si el equipo de *testing* es involucrado en etapas tempranas, se podrá cuestionar la definición del requerimiento, y de esa forma incluso mejorar la documentación o la definición misma del requisito. Por ejemplo, si plantean hacer una funcionalidad para gestionar los usuarios de una biblioteca, al leer la documentación, el equipo de *testers* podrá realizar algunas preguntas para clarificar puntos que no estén determinados, que pueden dar lugar a ambigüedad. En el ejemplo, al usuario se le pide la cédula de identidad, y quien trabaje como *tester* quizá pregunte qué pasa en la situación de que una persona extranjera sea usuaria y no tenga documento de identidad del país en que se dé la situación. Si hay algo ambiguo en la documentación, hay riesgos de que usuarios o clientes del sistema hayan asumido algo y que luego el equipo de desarrollo asuma algo distinto. Entonces se genera un problema, una expectativa no satisfecha, incluso un *bug*. Siguiendo el ejemplo, quizá quien posea la titularidad de la biblioteca asumió que podrán usarse distintos tipos de documentos de identidad, según el país de procedencia de las personas que acudan a ella, o incluso quizá nunca pensó en la posibilidad de que fueran extranjeras. Esto hace que analicemos esa situación y tomemos una decisión de cómo diseñar el sistema para que se adapte mejor a usuarios y necesidades.

Esto es una parte importante del trabajo de *tester*, ese pensamiento “fuera de la caja” que permita descubrir puntos ciegos. Una vez que se cuenta con la descripción del requerimiento, se comienza a cuestionar, a pensar cómo podría probar esa funcionalidad, cómo alguien podría llegar a usarla, qué sería un resultado esperado para determinadas situaciones, etc.

Según la metodología usada y otro montón de factores, esas pruebas que se van pensando podrán ser documentadas en planillas, documentos, herramientas o mapas mentales (que veremos más adelante). Uno de los formatos más clásicos es el de *caso de prueba*, pero hay más.

Según la complejidad de lo que estemos probando, podrá haber miles de pruebas a ejecutar. ¡Y no estamos exagerando! Lo fundamental es determinar qué pruebas tendrán más probabilidad de encontrar errores que otras, y qué partes del sistema son más críticas para el negocio que otras. Considerando todo esto, es crucial priorizar nuestro trabajo para poder maximizar los resultados. Entonces, no solo es importante saber qué probar y cómo, es igual de importante saber qué **no** probar.

Cuando el equipo de desarrollo deja disponible una primera versión de la funcionalidad, entonces, ya podemos comenzar a ejecutar nuestras pruebas. Típicamente, esta funcionalidad estará en un entorno de pruebas específico para ir probando las cosas de manera temprana, interna al equipo de desarrollo, sin que usuarios accedan a estos entornos. Esto hace que el contexto sea seguro para experimentar y aprender.

Las personas que prueban dentro del equipo de desarrollo ejecutarán todas sus pruebas; intentarán así entender el estado de calidad de esa primera versión de la funcionalidad, y para esto también estarán en búsqueda de defectos, que se irán reportando.

Reportar errores

Cuando *testing* encuentra defectos, *bugs*, errores o incluso sugerencias de mejora, serán reportados en alguna plataforma, para que puedan ser corregidos por el equipo de desarrollo. La mayoría de las veces se contará con una herramienta para la gestión de los incidentes, como para poder documentar los errores encontrados y darles seguimiento. Ahí se podrá ver si ya se resolvieron, si se está trabajando en el error, o si aún no se ha comenzado.

Cuando se reportan errores, se suele describir el error, los pasos que se ejecutaron y los datos usados para poder reproducirlo. En muchas ocasiones se presentan capturas de pantalla o videos para facilitar a desarrollo que pueda reproducir el incidente, ya que eso por un lado le permitirá entender el problema, y a su vez, luego de encontrar una solución, le va a permitir verificar que realmente se haya solucionado.

Retest

Cuando desarrollo indica que el problema está resuelto y que ya dejó disponible la solución en el entorno de pruebas, *testing* toma la posta otra vez y hace algo que se

le llama *retest*, o sea, vuelve a probar la funcionalidad para verificar que el arreglo realmente soluciona el problema.

Ya sea que se verifique que está resuelto o que se observe que el problema persiste, se le indica el resultado del retest a desarrollo, como para que pueda continuar según haga falta.

Pruebas de regresión

Además del retest, luego de que se implementan cambios, es importante verificar que todo lo que antes funcionaba, en forma previa a esos cambios, sigue funcionando adecuadamente. Como el *software* es complejo y unas funcionalidades dependen de otras, están interconectadas, cuando algo se cambia para una funcionalidad, estos cambios pueden afectar a otras áreas del sistema. Esto significa que, cuando se agrega o cambia funcionalidad, existe el riesgo de que algo más deje de funcionar.

Para esto, están las pruebas de regresión. Generalmente se cuenta con un grupo reducido de pruebas que nos permiten validar que al menos los flujos más importantes de la aplicación aún estén funcionando.

Liberar a producción

Luego de estar de acuerdo sobre que la funcionalidad esté implementada con la calidad adecuada, que pueda colmar las expectativas de clientes y que ningún otro componente haya sido afectado, se le hará llegar esa versión al usuario. En algunas ocasiones, esto se hace de manera individual, funcionalidad a funcionalidad, y en otras, se liberan paquetes que cuentan con varias funcionalidades, que acumulan lo trabajado en una semana, dos semanas o incluso más tiempo.

Muchas veces, luego de que se pasa a producción, se hace una prueba para verificar que todo funcione adecuadamente en ese entorno, que al fin de cuentas es el entorno más importante, ya que es el que utilizan los usuarios.

Reportes

De nuevo, dependiendo de la metodología del equipo, será costumbre hacer más o menos reportes con más o menos información. Es importante dar visibilidad del trabajo realizado, entonces, muchas veces se plantean reportes que resumen el progreso del trabajo, el estado de calidad del producto, *bugs* encontrados, resueltos, en progreso, y se indica la severidad e impacto de cada incidente, entre otras métricas.

Según el contexto, estos informes quizá deban ser extensos y detallados, o en otras situaciones serán breves o incluso, en lugar de reportes escritos, serán presentados verbalmente.

Reuniones

Se suele pensar que en desarrollo de *software* las personas “hablan con las computadoras”, pero en realidad hay un alto porcentaje del tiempo en que se está interactuando con otras personas. Estamos hablando de trabajo en equipo, donde es fundamental estar en sintonía. Además, se está trabajando en desarrollar algo complejo (*software*) para atender las necesidades de los usuarios, con lo cual habrá que asegurarse de colmar sus expectativas. Todo esto lleva a que lo típico es que haya muchas reuniones.

Veamos algunas de las más típicas:

- Reuniones de planificación: básicamente se planifica el trabajo y se distribuyen responsabilidades entre los distintos miembros del equipo.

- Reuniones de seguimiento o revisión: sirven para revisar que se está siguiendo el plan o si hace falta comunicar desvíos. Estas reuniones podrán ser diarias, semanales o incluso de menor frecuencia, dependiendo de qué personas y roles estén involucrados.
- Reuniones de requerimientos: se busca entender más los requerimientos, con lo cual a veces se cuenta con reuniones con los usuarios o con los clientes (quienes quieren desarrollar el sistema para sus usuarios) o con personas que conocen bien el negocio o el contexto de los usuarios.
- Reuniones técnicas: muchas veces es necesario discutir la forma de solucionar los problemas y requerimientos, con lo cual se llevan a cabo reuniones para discutir cuál es el enfoque técnico que se les dará.
- Reuniones de inicio o cierre de proyectos.
- Reuniones de retrospectiva, *feedback*, aprendizaje.

Podrá haber más o menos reuniones, pero queríamos dar un panorama de qué tipo de actividades suele participar quien está en *testing*.

¿Qué se necesita para poder probar?

Luego de haber visto el panorama global muy en alto nivel, hay varias cosas que queda más claro que se necesita para poder ser *tester*. Hay que pensar qué probar y qué no, ver cómo ejecutar esas pruebas, ordenar y priorizar. Para eso, entender el negocio y a nuestros usuarios, como para decidir dónde enfocarnos y qué probar primero. Es importante dar visibilidad a nuestro trabajo, poder colaborar y trabajar en equipo. Es necesario pensar “fuera de la caja”, aportar valor desde el inicio (al pensar un requerimiento) hasta el final (cuando se pasa a producción). Siempre con los usuarios en vista, siempre pensando en cómo mejorar la experiencia de los usuarios, dando *feedback* oportuno y de calidad.

¿Cómo se sienten para afrontar el desafío? No se preocupen, aún hay mucho para leer.

Conceptos introductorios de *testing*

En este apartado comenzaremos con definiciones básicas. Si estás leyendo el libro, significa que te interesa el *testing* y probablemente conozcas todas estas definiciones, o quizá no. De cualquier modo, si en algún momento necesitas repasar algún concepto, puedes referirte a esta parte del libro.

Caso de prueba

¿De qué hablamos cuando nos referimos a un *caso de prueba*? Veamos algunas definiciones, pero no sin antes aclarar que no todo *testing* implica definir, documentar y basarse en casos de prueba, pero creemos que es un concepto que va a aparecer tantas veces que lo mejor es que lo conozcamos de arranque:

De acuerdo al Estándar IEEE 610 (1990) un caso de prueba se define como:

Un conjunto de entradas de prueba, condiciones de ejecución y resultados esperados desarrollados con un objetivo particular, tal como el de ejercitar un camino en particular de un programa o el verificar que cumple con un requerimiento específico.

Brian Marick utiliza un término relacionado para describir un caso de prueba que está ligeramente documentado, referido como *idea de prueba*¹⁷:

Una idea de prueba es una breve declaración de algo que debería ser probado. Por ejemplo, si se está probando la función de la raíz cuadrada, una idea de prueba sería el “probar un número que sea menor que cero”. La idea es chequear si el código maneja un caso de error.

¹⁷ *Test idea*, Brian Marick: <<http://www.exampler.com/testing-com/tools.html>>

Por último, una definición provista por Cem Kaner (2003):

Un caso de prueba es una pregunta que se le hace al programa. La intención de ejecutar un caso de prueba es la de obtener información, por ejemplo sea que el programa va a pasar o fallar la prueba. Puede o no contar con gran detalle en cuanto a procedimiento, en tanto que sea clara cuál es la idea de la prueba y cómo se debe aplicar esa idea a algunos aspectos específicos (característica, por ejemplo) del producto.

Digamos que el caso de prueba es “la personalidad más famosa” en el mundo del *testing* y, como verán en alguna literatura, para algunos tiene mala fama. Más adelante, veremos que hay ciertos enfoques de *testing* que promueven la libertad, la creatividad, y para eso se oponen a tener casos de prueba definidos, y de ahí que los que prefieren este enfoque ven al caso de prueba con malos ojos, más que nada porque no deberíamos limitar el *testing* solo a este enfoque. Teniendo esto en mente, veamos un poco más de cómo se componen.

El caso de prueba debe incluir varios elementos en su descripción, entre los que queremos destacar, se encuentran:

- Descripción, objetivo, ¿por qué queremos probar este caso?
- Flujo: secuencia de pasos a ejecutar.
- Datos entrada.
- Estado inicial.
- Valor de respuesta esperado.
- Estado final esperado.

Lo más importante que define a un caso de prueba es: **el flujo**, o sea, la serie de pasos a ejecutar sobre el sistema; **los datos de entrada** (ya sean entradas proporcionadas por el usuario al momento de ejecutar o el propio estado de la aplicación) y por último **las salidas esperadas**, el oráculo, lo que define si el resultado fue positivo o negativo.

Oráculo

Hablar de *caso de prueba* nos lleva a pasar a hablar del concepto de *oráculo*. Básicamente, es el mecanismo, ya sea manual o automático, de **verificar si el comportamiento del sistema es el deseado o no**. Para esto, el oráculo deberá comparar el valor esperado contra valor obtenido, el estado final esperado con el estado final alcanzado, el tiempo de respuesta aceptable con el tiempo de respuesta obtenido, etc.

En este punto es interesante reflexionar sobre algo a lo que se llama la **Paradoja del pesticida**: los insectos (*bugs*, refiriéndose a fallos, ¡nunca vino tan bien una traducción literal!) que sobreviven a un pesticida se hacen más fuertes y resistentes a ese pesticida. O sea, si diseñamos un conjunto de pruebas, probablemente ciertos *bugs* sobrevivan. Si luego diseñamos una técnica más completa y, llamémosle, *exhaustiva*, entonces encontraremos más *bugs*, pero otros seguirán sobreviviendo. Al fin de cuentas, los que van quedando son los más duros de matar y se van haciendo resistentes a los distintos pesticidas.

Cobertura de pruebas

Cobertura o cubrimiento, no está claro. Suponemos que depende del país, pero siempre se entiende, ya se use una u otra. En inglés, *coverage*.

Básicamente, es una medida de calidad de las pruebas. Se define cierto tipo de entidades sobre el sistema y luego se intenta cubrirlas con las pruebas. Es una forma de indicar cuándo probamos suficiente o para tomar ideas de qué otra cosa probar (pensando en aumentar la cobertura elegida). Por ejemplo, si estamos probando un sistema que tiene 5 módulos, para considerar una cobertura muy alto nivel, podremos indicar cuántos módulos cubrimos con nuestras pruebas, indicando algo como “ya probamos 4 de los 5 módulos”, esa sería nuestra cobertura en ese momento. Podemos ir más al detalle indicando cuántas funcionalidades en cada módulo o incluso pensar en cuántas líneas de código cubrimos (en este último

caso, una línea de código es lo que consideramos como entidad). En este ejemplo, podríamos indicar, por ejemplo, que “nuestras pruebas cubren el 85% de nuestro código”.

Para verlo aún más simple, podríamos decir que la cobertura es como cuando barremos la casa. Siempre se nos olvida el cuarto, eso es que en nuestro barrido no estamos cubriendo el cuarto. Mide la calidad de nuestro barrido bajo un determinado criterio y a su vez nos da una medida para saber cuándo tenemos que terminar de barrer: cuando hayamos cubierto cada habitación, por ejemplo.

Ahora, lograr el 100% de cobertura con ese criterio, ¿indica que la casa está limpia?

¡No!, porque la cocina y el comedor ¡ni los miramos! Entonces, ¡ojo! Debemos manejar el concepto con cuidado. Tener cierto nivel de cobertura es un indicador de la calidad de las pruebas, pero nunca es un indicador de la calidad del sistema, por ejemplo, ni nos garantizará que está todo probado.

¿Entonces para qué nos sirve?

- Medida de calidad de cómo barremos.
- Nos indica cuándo parar de barrer.
- Nos sugiere qué más barrer.

Unos criterios pueden ser más fuertes que otros, entonces, el conocerlos nos puede dar un indicador de qué tan profundas son las pruebas, cuándo aplicar uno y cuándo otro. Se dice que “un criterio A subsume a otro criterio B” cuando cualquier conjunto de casos de prueba que cubre el criterio A también cubre el criterio B.

- Criterio 1: barrer cada habitación.
- Criterio 2: barrer cada pieza (habitaciones, comedor, cocina, baño, etc.).
- Criterio 3: barrer cada pieza, incluso en las esquinas, porque ahí hay más posibilidades de que se acumule suciedad.

El criterio 3 subsume al criterio 2, el cual subsume al criterio 1 (y la relación es transitiva, con lo cual el criterio 3 subsume al criterio 1).

La analogía es evidente.

Técnicas de diseño de casos de prueba

Existen distintas técnicas de diseño de casos de prueba, que permiten seleccionar la menor cantidad de casos con mayor probabilidad de encontrar fallas en el sistema. Por este motivo, estos casos se consideran los más interesantes para ejecutar, ya que el *testing* exhaustivo no solo es imposible de ejecutar en un tiempo acotado, sino que también es muy caro e ineficiente. Es así que se vuelve necesario seleccionar de una forma inteligente los valores de entrada que tengan más posibilidades de descubrir un error. Para esto, el diseño de pruebas se basa en técnicas bien conocidas y utilizadas, tales como particiones de equivalencia, valores límites, combinaciones por pares, tablas de decisión o máquinas de estado. Por ejemplo, probar valores límites, basado en que hay siempre probabilidad de que no se estén controlando estas situaciones por errores en la programación, como por ejemplo al comparar dos variables se cometa el error de usar un comparador de *menor* en lugar de un *menor igual*. También existen técnicas más avanzadas, como la que implica el uso de máquinas de estado. En este libro veremos unas cuantas técnicas para tenerlas siempre en el bolsillo al momento de diseñar pruebas.

Caja blanca y caja negra

La clasificación más importante y más difundida de técnicas de diseño de casos de prueba está basada en la información utilizada como entrada. Si utilizamos información interna del sistema que estamos probando, tal como el código, esquema de base de datos, etc., entonces se dice que estamos siguiendo una estrategia de **caja blanca**. Si en cambio nos basamos únicamente en la observación de entradas y salidas del sistema, estamos siguiendo un enfoque de **caja negra**. Efectivamente, ese caso sería como considerar que el sistema es una caja a la cual no podemos mirar su interior. Por esta asimilación es también que a las técnicas de caja blanca a veces se les dice técnicas de *caja transparente*,

haciendo mejor referencia a que la idea es poder mirar qué hay dentro de esa caja que estamos probando.

Podríamos decir que con caja blanca nos preocupamos por lo que pasa dentro de la caja y con caja negra nos preocupamos por lo que pasa fuera de ella. Muchas veces el límite no está claro o tal vez estamos siguiendo un enfoque de caja negra, pero como sabemos algo de lo que sucede dentro entonces aprovechamos esa información. También hay quienes hablan de *caja gris*, cuando se combinan ambos enfoques.

Lo importante es que puede haber una diferencia en el alcance, la forma de hacernos las preguntas, la información y los objetivos de las pruebas que diseñemos.

Caso de prueba abstracto y específico

Este tipo de clasificación también es muy difundida y particularmente útil. Más que nada, porque nos habla de la especificidad con la que está detallado el caso de prueba.

Un **caso de prueba abstracto** se caracteriza por no tener determinados los valores para las entradas y salidas esperadas. Se utilizan variables y se describen con operadores lógicos ciertas propiedades que deben cumplir (por ejemplo, *edad* > 18 o *nombre válido*). Entonces, la entrada concreta no está determinada.

Un **caso de prueba específico** (o concreto) es una instancia de un caso de prueba abstracto, en la que se determinan valores específicos para cada variable de entrada y para cada salida esperada (en el ejemplo anterior, no alcanza con decir que queremos usar una edad mayor a 18, deberíamos indicar que queremos usar, por ejemplo, el valor 23 para la variable edad).

Cada caso de prueba abstracto puede ser instanciado con distintos valores. Esto significa que al momento de ser ejecutado (o diseñado a bajo nivel) tendrá un conjunto de casos de prueba específicos donde se asigne un valor concreto a cada

variable (tanto de entrada como de salida). Esto siempre de acuerdo con las propiedades y restricciones lógicas que tiene determinadas a nivel abstracto.

Pruebas Dirigidas por Datos

La clasificación anterior nos da pie para hablar de una técnica de *testing* que es muy útil y que puede que nombremos en varias situaciones: Pruebas Dirigidas por Datos o, del inglés, bien conocida como *Data-Driven Testing*.

Es una técnica para construir casos de prueba basándose en los datos de entrada y separando el flujo que se toma en la aplicación. O sea, por un lado se representa el flujo (la serie de pasos para ejecutar el caso de prueba) y por otro lado se almacenan los datos de entrada y salida esperados. Esto permite agregar nuevos casos de prueba fácilmente, ingresando simplemente nuevos datos de entrada y de salida esperados, que sirvan para ejecutar el mismo flujo.

Veamos un ejemplo aplicado al *login* de un sistema *web*. Por un lado, definiríamos el flujo del caso de prueba, el cual nos indica que tenemos que ir a la URL de la aplicación, luego ingresar un nombre de usuario, un *password*, dar OK y nos aparecería un mensaje que diga *Bienvenid@*. Por otro lado, tendríamos una fuente de datos en donde especificaríamos distintas combinaciones de nombres de usuario, *password* y mensaje de respuesta esperado. El día de hoy se nos ocurre poner un usuario válido, *password* válido y mensaje de bienvenida; mañana agregamos otro que valide que ante un *password* inválido nos dé un mensaje de error correspondiente y esto lo hacemos simplemente agregando una línea de datos, sin necesidad de definir otro caso de prueba distinto.

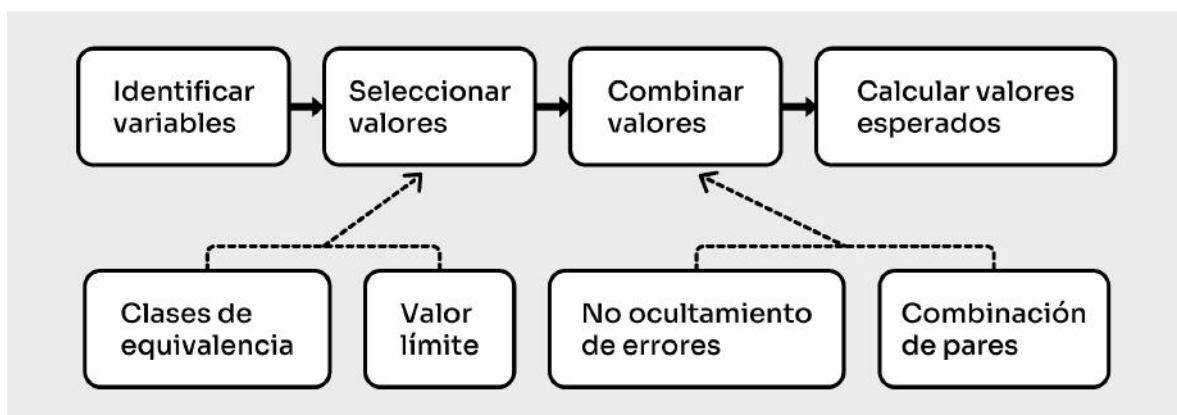
Entonces, el flujo de la aplicación se está definiendo con **casos de prueba abstractos** que, al momento de ser ejecutados con un juego específico de datos, se estarían convirtiendo en **casos de prueba específicos**. De ahí el vínculo entre *Data-Driven Testing* y las definiciones de casos de prueba abstracto y específico.

Diseño básico de casos de prueba

Ahora que tenemos algunas ideas sobre **qué** es un caso de prueba y **cómo** nos pueden ayudar en nuestras tareas, presentaremos algunas técnicas de diseño que son bastante genéricas y nos vendrán bien para muchas situaciones. Estas no son las únicas, sino que son las básicas sobre las que podemos comenzar a construir nuestro conjunto de pruebas inicial. Luego, podremos ir mejorando el cubrimiento del sistema incorporando otras técnicas o alimentando la que ya estamos usando, personalizándola de acuerdo a nuestros gustos, criterios y formas de pensar. Esta personalización debe hacerse ajustándose a la aplicación sobre la que trabajemos.

Siempre que sea posible, vamos a seguir la estrategia de pruebas dirigidas por datos. Entonces, lo primero que haremos será determinar el flujo del caso de prueba, que, básicamente, será seguir el flujo de la funcionalidad. Más adelante, veremos estrategias para determinar distintos flujos a probar, pero por ahora pensemos en que queremos probar un determinado flujo y vamos a diseñar los datos de prueba.

En la siguiente figura presentamos un posible esquema de trabajo propuesto para diseñar datos de prueba para nuestra funcionalidad bajo prueba. Digamos que es una forma ordenada y cuasi-metodológica de comenzar.



Proceso básico de diseño de pruebas

Resumiendo, primero se identifican las variables que están en juego en la funcionalidad; luego, para cada variable se diseñan valores interesantes, para lo cual es útil aplicar técnicas de partición en clases de equivalencia y valores límites; por último, estos valores deben combinarse de alguna forma, porque seguramente el producto cartesiano (todas las combinaciones posibles) nos daría una cantidad de casos de prueba muy grande y, quizá, muchos de esos casos tendrían “poco valor agregado”. Veamos cada parte con mayor profundidad.

Identificar variables

El comportamiento esperado de cada funcionalidad a probar es afectado por una serie de variables que deben ser identificadas para el diseño de pruebas. Tal como indica el nombre, estamos hablando de variables, de cualquier entrada que, al cambiar su valor, hace que **cambie el comportamiento o resultado** del sistema bajo pruebas.

¿Qué puede conformar las variables de una funcionalidad?:

- Entradas del usuario por pantalla.
- Parámetros del sistema (en archivos de configuración, parámetros de invocación del programa, tablas de parámetros, etc.).
- Estado de la base de datos (existencia –o no– de registros con determinados valores o propiedades).
- Versiones de las tecnologías usadas (por ejemplo, versión del SQL Server, versión del Tomcat, versión del sistema operativo).
- Etc.

Existen muchas entradas de un sistema que tal vez no afecten el comportamiento o resultado de lo que estamos ejecutando. Lo mismo para el estado inicial de la base de datos, tal vez existen cientos de registros, muchas tablas, etc., pero no todas afectan la operación que estamos probando. Se debe identificar qué variables son las que nos interesan, para focalizar el diseño de las pruebas y de los datos de

prueba considerando estas variables, de lo contrario, estaremos diseñando y ejecutando más pruebas que quizá no aporten valor al conjunto de pruebas.

Veamos un ejemplo simple de una aplicación que permite ingresar facturas (*Invoices*) simplemente ingresando el cliente al que se le vende, fecha y líneas de producto, donde se indica cada producto y la cantidad de unidades que compra. En la siguiente figura se ve la pantalla *web* para el ingreso de facturas. El sistema verifica que los valores ingresados en los campos *Client Name* y *Product Name* existan en la base de datos. En el caso del producto, al ingresar un nombre, carga en la fila los valores para *Stock* y *Price*. Con el valor del precio y al ingresar la cantidad de unidades deseada, se calcula el *Line Amount* de esa fila.

Invoice

Id 0

Date 01/17/13

Client Name

Line Id	Product Name	Stock	Price	Line Quantity	Line Amount
0		0	0.00	0	0.00
0		0	0.00	0	0.00
0		0	0.00	0	0.00
0		0	0.00	0	0.00
0		0	0.00	0	0.00

[New row]

Sub Total 0.00

Pantalla para ingresar facturas en un sistema de ejemplo

Claramente, el nombre de cliente es una variable, así como el producto y la cantidad del producto que se quiere facturar. Otras menos evidentes tal vez podrían ser:

- *Stock y precio del producto*: a pesar de que son valores que no ingresa directamente el usuario, es interesante considerar qué pasa cuando se intenta comprar más cantidad de la que hay en *stock* o si el precio es negativo o inválido. Acá estamos considerando valores ya existentes en la base de datos y los controlamos con la variable *Producto*, o sea, para variar estos valores necesitaremos ingresar productos con las características deseadas y luego seleccionarlos.
- *Cantidad de líneas de la factura*: esta es una variable interesante generalmente en toda lista, la cantidad de elementos, donde vamos a querer ver qué pasa cuando intentamos crear una factura sin ninguna línea de producto o con una cantidad muy grande.
- *Subtotal de la línea*: hay variables que son cálculos de otras, como en este caso, que se trata de la multiplicación de la cantidad por el precio del producto. Sería interesante ver qué pasa con valores muy grandes, verificando que no se produzca un *overflow*¹⁸ en la operación.

Analizando más la especificación del sistema, imaginen que nos encontramos con que se registra el balance de cliente, con lo cual, si este tiene un balance negativo, no se le habilitará el pago a crédito. Por esto, es interesante considerar la variable *balance* asociada al cliente ingresado. Esto también corresponde a un valor de la base de datos.

Seleccionar valores interesantes

Una vez que se identifican las variables en juego en la funcionalidad bajo pruebas, para cada una hay que seleccionar los valores que se les asignarán en las pruebas. Para cada variable se decidirá utilizar distintos datos, los cuales sean interesantes desde el punto de vista del *testing*.

¹⁸ *Overflow (desbordamiento)*: Situación en la que un cálculo o una operación produce un resultado que excede la capacidad del sistema para manejarlo. Puede ocurrir en la aritmética de los computadores o en la gestión de memoria, y lleva a resultados incorrectos o errores en el programa.

Para diseñar los conjuntos de datos de prueba, se analizan las reglas del negocio, los tipos de datos y los cálculos a realizar, para poder determinar primero que nada las **clases de equivalencia**. En términos de *testing*, se considera que un grupo de valores pertenece a una misma clase de equivalencias si deben producir un comportamiento similar en el sistema. Considerando eso, podríamos decir que, al elegir un representante cualquiera de cada clase de equivalencia, será *suficiente* para probar el total de las entradas.

¿Cómo se identifican las distintas clases de equivalencia? Pues, pensando en opciones *significativamente diferentes*, o sea, que generan distintos comportamientos.

Por ejemplo, si tenemos una variable para ingresar un identificador que acepta cadenas de entre 5 y 10 caracteres podremos pensar en *Fede*, que es muy corto, por lo que debe dar un error; *Federico*, que es una entrada válida, y *Federico Toledo*, que es muy largo, por lo que debe dar error. En cambio, *Federico* y *Andrea* son ambas válidas, no son *significativamente diferentes*, por lo que corresponden a la misma clase de equivalencia, ambas deberían generar el mismo comportamiento en el sistema.

Podemos distinguir opciones significativamente diferentes cuando:

- Disparan distinto flujo sobre el sistema (por ejemplo, al ingresar un valor inválido me lleva a otra pantalla distinta).
- Dispara un mensaje de error diferente (no es lo mismo ingresar un *password* corto que uno vacío).
- Cambia la apariencia de la interfaz del usuario o el contenido del *form* (al seleccionar pago con tarjeta de crédito aparecen los campos para ingresar el número de tarjeta, titular y fecha de expiración, pero si se selecciona pago por transferencia, entonces, aparecen los campos para ingresar información de la cuenta bancaria).
- Cambian las opciones disponibles para otras variables (al seleccionar un país distinto, se pueden seleccionar sus estados o provincias).
- Si dispara distintas condiciones para una regla de negocio (si se ingresa una edad, será distinto el comportamiento si es menor o mayor de edad).

- Valores por defecto o cambiados (en ocasiones, el sistema sugiere datos que se habían ingresado previamente para un cliente, como por ejemplo su dirección, pero si se cambian en el formulario de entrada entonces el sistema actualizará ese dato, con lo cual habrá un comportamiento extra).
- Cambios de comportamiento por distintas configuraciones o distintos países (formato de números, formatos de fecha, moneda, etc.).

¿Qué más se les ocurre? Seguro que hay más, estos fueron ejemplos para mostrar cómo analizar los casos significativamente diferentes.

Para seleccionar los representantes de cada clase, lo primero es definir cuáles son clases válidas e inválidas. Por ejemplo, si estamos hablando de un campo de entrada que es para indicar la edad de una persona y se sabe que el comportamiento cambia respecto a si se trata de un menor de edad o se trata de un mayor de 18 años, entonces, las clases que se pueden definir son:

- Clase 1= (-infinito, 0)
- Clase 2= [0, 18)
- Clase 3= [18, 100]
- Clase 4= (100, infinito)

Sabemos que las clases 1 y 4 son inválidas porque no habrá edades mayores de 100 (típicamente) y tampoco son válidos los números negativos. Por otra parte, se podría definir una quinta clase, también inválida, que esté compuesta por cualquier carácter no numérico, o incluso el valor vacío (un *string* de largo 0). Luego, las clases 2 y 3 son válidas y será interesante seleccionar representantes de cada una, como podría ser el valor 15 para la clase 2 y el valor 45 para la clase 3.

Prestemos atención a los símbolos utilizados. Los paréntesis curvos no incluyen el valor en el límite, y los rectos sí. Esto significa que el 18 está incluido en la clase 3 y no en la 2.

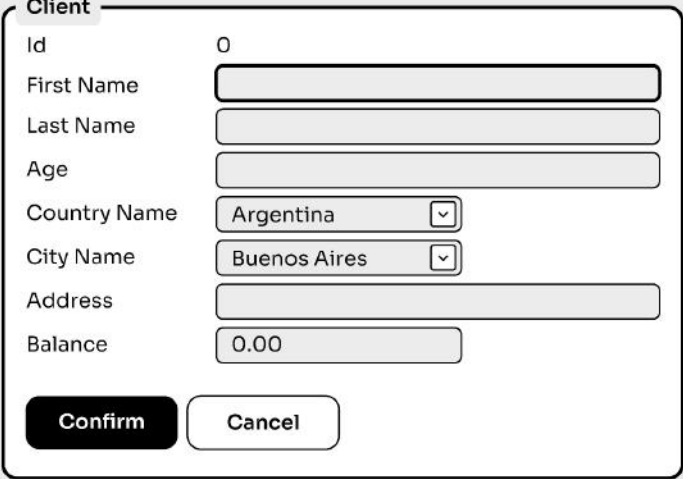
Considerando esto, pensemos que serán también interesantes los **valores límites** de las clases de equivalencia, es decir, para la clase 2 el valor 0 y el 17 (límites incluidos en la clase) y 18 (límite por fuera de la clase). Probar con estos valores tiene sentido, ya que un error muy común en los programadores es el de utilizar

una comparación de mayor en lugar de mayor o igual, o similares. Este tipo de errores se puede verificar solo con los valores que están al borde de la condición.

Observemos aquí que comienza a tener más sentido la separación entre caso de prueba abstracto y caso de prueba específico. El caso de prueba abstracto podría estar definido haciendo referencia a qué clase de equivalencia se utiliza en la prueba y luego se instancia con un representante de esa clase de equivalencia.

Veamos un ejemplo.

La funcionalidad de dar de alta un cliente se realiza con la pantalla que muestra la siguiente figura.



Formulario de creación de cliente (Client) con los siguientes campos:

- Id:** 0
- First Name:** Campo de texto vacío.
- Last Name:** Campo de texto vacío.
- Age:** Campo de texto vacío.
- Country Name:** Combobox con "Argentina" seleccionado.
- City Name:** Combobox con "Buenos Aires" seleccionado.
- Address:** Campo de texto vacío.
- Balance:** 0.00

Botones: Confirm (negro) y Cancel (gris).

Pantalla para crear un cliente en el sistema de ejemplo

El identificador *Id* es autogenerado al confirmar la creación. Los campos *First name* y *Last name* se guardan en campos del tipo Char(30) en la base de datos (o sea, *strings* de hasta 30 caracteres). El campo *Address* está definido como Char(100). Tanto *Country Name* como *City Name* se presentan en *comboboxes*

cargados con los valores válidos en la base de datos. Cada cliente se trata en forma distinta según si es del mismo país o si es extranjero (por impuestos que se deben aplicar). Solo se pueden dar de alta clientes mayores de edad.

Comentario al margen: ¿acá estamos aplicando caja blanca o caja negra? El enfoque general es de caja negra, pero al mismo tiempo estamos preocupándonos por el tipo de dato definido en la base de datos para almacenar los valores, lo cual es bastante interno a la caja. Podríamos decir que es una caja gris, quizá.

Siguiendo con el ejemplo, las variables con sus clases válidas e inválidas y sus valores interesantes se podrían diseñar como muestra la siguiente tabla.

Variable	Clases de equivalencia	Válida o inválida	Valores interesantes
First Name	Hasta 30 caracteres	Válida	“Federico”
	Más de 30	Inválida	<i>String</i> de 31 caracteres
	Vacía	Inválida	“”
Last Name	Hasta 30 caracteres	Válida	“Toledo”
	Más de 30	Inválida	<i>String</i> de 31 caracteres
	Vacía	Inválida	“”
Age	$0 \leq x < 18$	Inválida	0, 1, 5, 17
	$18 \leq x \leq 100$	Válida	18, 30, 99, 100
	Negativos	Inválida	-1
	Caracteres no numéricos	Inválida	“asdf”
Country	Cualquiera extranjero	Válida	España
	Local	Válida	Uruguay
City	Cualquiera	Válida	(cualquiera seleccionable)
Address	Hasta 100 Caracteres	Válida	“Bulevar Artigas, 1”
	Más de 100	Inválida	<i>String</i> de 101 caracteres
	Vacía	Inválida	“”
Balance			

Diseño de datos de prueba para ejemplo

¡Ejercicio! Completar la tabla diseñando clases de equivalencia (indicando si es válida o inválida, y diseñando valores interesantes para cada una) para la variable *Balance*, considerando que puede ser positiva, negativa o nula. En cada caso el comportamiento del sistema será distinto. El sistema no debe permitir que un cliente tenga una deuda mayor a 500 dólares, ni más de 100 a favor.

Combinación de valores

Cuando diseñamos casos de prueba, se utilizan datos para sus entradas y salidas esperadas y por lo tanto también diseñamos datos de prueba para cada variable que está en juego (cuando decimos *diseñar datos*, nos referimos a elegirlos, definir cuáles vamos a usar). Cada caso de prueba se ejecutará con distintos juegos de datos, pero ¿es necesario utilizar todas las combinaciones posibles? Con casos muy simples, que manejan pocas variables y pocos valores, ya podemos observar que la cantidad de ejecuciones que necesitamos serían muchas. Por esto es que tenemos que intentar encontrar las combinaciones de datos que tienen más valor para el *testing*, o sea, las que tienen más probabilidad de encontrar errores.

Las **técnicas de Testing Combinatorio** permiten seleccionar una cantidad acotada de datos de prueba, siguiendo teorías de errores que indican cómo combinar los datos para aumentar esa probabilidad de encontrar errores, es decir, se busca obtener el conjunto de datos más eficiente posible (*eficiente*, entendido como encontrar la mayor cantidad de errores con la menor cantidad de ejecuciones).

Combinación por pares

Una de las técnicas de combinación de datos más usada es *All-pairs*, o *Pairwise*, o sea, *todos los pares*, y justamente, lo que se hace es intentar utilizar todos los pares posibles. En otras palabras, si vinculamos esto con el concepto de *cobertura*, lo que plantea esta técnica es que cubre todos los pares de las combinaciones de datos.

Para entenderlo mejor, veamos un ejemplo. Imaginen que tenemos 3 variables para un caso de prueba sencillo y se seleccionaron valores interesantes para cada

variable, como se detalla a continuación (tomado de un proyecto real, pero despersonalizado para no involucrar a nadie).

Funcionalidad bajo pruebas: Solicitud de servicio por parte de un cliente.

Variables en juego y conjuntos de valores interesantes:

- Canal: {Presencial, Telefónico, *email*}
- Prioridad: {Urgente, Alta, Media, Baja}
- Tipo de Servicio: {Adhesión Tarjeta Débito, Adhesión Tarjeta Crédito, Adhesión *e-commerce*}

El caso de prueba <Presencial, Media, Adhesión Tarjeta Débito> cubre los pares <Presencial, Media>, <Presencial, Adhesión Tarjeta Débito> y <Media, Adhesión Tarjeta Débito>. **La técnica *Pairwise Testing* implica diseñar un conjunto mínimo de pruebas para cubrir todos los pares posibles.** Si intentamos hacer esto manualmente, sería una tarea muy costosa.

Existen muchas herramientas¹⁹ que aplican distintas estrategias para conseguir combinaciones de datos que cubran todos los pares (e incluso no solo pares, sino tripletas o combinaciones de “n” elementos). En la primera edición del libro mostramos un ejemplo con la herramienta diseñada por Macario Polo y Beatriz Pérez Lamancha llamada *CTWeb*, desarrollada en la Universidad de Castilla-La Mancha, la cual ya no se encuentra disponible.

Sigamos ahora el ejemplo con la herramienta *PICT online*²⁰, que básicamente es la herramienta *PICT* de Microsoft²¹ pero con una interfaz *web*.

Primero, ingresamos al sitio y definimos las variables con sus valores interesantes según el formato presentado y presionamos el botón “Generate=>”.

¹⁹*Pairwise testing*, sitio donde se puede encontrar un listado de herramientas para aplicar esta técnica: <<http://www.pairwise.org/tools.html>>

²⁰*PICT online*: <<https://pairwise.yuuniworks.com>>

²¹ *PICT* de Microsoft: <<https://github.com/microsoft/pict>>

Pairwise Pict Online
Buy me a coffee
Star
3

An online service that easily generates pair-wise test cases.
It's powered by [Microsoft Pict](#) under the hood.

```

#####
# Paste test factors here.
# Check the documents for more details.
# https://github.com/Microsoft/pict/blob/master/doc/pict.md
#####

Canal:      Presencial, Telefonico, e-mail
Prioridad:  Urgente, Alta, Media, Baja
Tipo de Servicio: Adhesion Tarjeta Debito, Adhesion Tarjeta
Credito, Adhesion e-commerce

if [Canal] = "e-mail" then [Prioridad] <> "Urgente";
if [Canal] = "e-mail" then [Prioridad] <> "Alta";

```

Download

Canal	Prioridad	Tipo de Servicio
e-mail	Baja	Adhesion Tarjeta Credito
Presencial	Media	Adhesion Tarjeta Debito
Telefonico	Urgente	Adhesion e-commerce
Presencial	Alta	Adhesion e-commerce
Telefonico	Media	Adhesion Tarjeta Credito
Telefonico	Baja	Adhesion Tarjeta Debito
Telefonico	Alta	Adhesion Tarjeta Debito
Presencial	Baja	Adhesion e-commerce
Presencial	Alta	Adhesion Tarjeta Credito
Presencial	Urgente	Adhesion Tarjeta Debito
e-mail	Media	Adhesion Tarjeta Debito
Telefonico	Urgente	Adhesion Tarjeta Credito
e-mail	Media	Adhesion e-commerce

Download

Pairwise Pict Online

Observemos que no se pueden usar tildes y que agregamos dos restricciones para indicar que los pares (*email*, Urgente) e (*email*, Alta) no los queremos, ya que no se pueden hacer solicitudes por correo electrónico de prioridad alta y urgente.

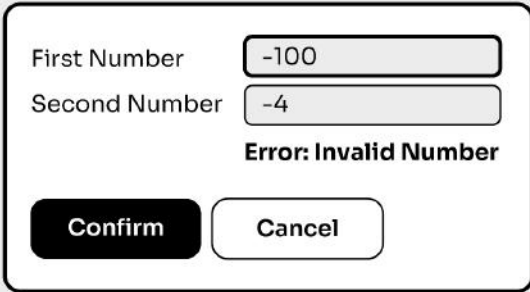
El resultado consta de 13 casos de prueba, los cuales cubren todos los pares válidos. Si decidiéramos hacer producto cartesiano con todos los valores de prueba interesantes tendríamos un total de $3 \times 4 \times 3 = 36$ casos de prueba. O sea, **nos estamos ahorrando la ejecución** de $36 - 13 = 23$ casos de prueba que, según la teoría de errores considerada, no agregarían valor significativo a nuestro conjunto de pruebas, ya que tienen menos probabilidades de encontrar errores.

Para aprovechar la herramienta, es muy importante seleccionar bien los valores interesantes a incluir en nuestras pruebas, así como luego determinar qué pares son inválidos o qué otras reglas de negocio considerar con respecto a los valores. No es lo mismo eliminar casos de prueba que tengan combinaciones inválidas luego de haberlos generado. Si hiciéramos eso (si eliminamos un caso de prueba generado por tener una combinación inválida), podemos estar a su vez eliminando otra combinación que sea interesante considerar y no se pruebe en ningún otro caso de prueba. Por esto es que tenemos que agregar las reglas y restricciones antes de generar los pares.

No enmascaramiento de errores

Otra técnica aplicable a la combinación de datos es la de *no enmascaramiento de errores*. Esto implica que en los conjuntos de datos de prueba se utilizará solo un valor de una clase inválida a la vez, pues si se utilizan dos se corre el peligro de no identificar qué valor inválido produce el error.

Para entenderlo bien, veamos un ejemplo muy simple: una pantalla que tiene dos entradas (ver la siguiente figura), ambas esperan valores enteros mayores o iguales a cero. Entonces, podemos distinguir como clases inválidas a los números negativos o a las entradas no numéricas. Esta técnica plantea no ingresar en un mismo caso de prueba un valor negativo en ambas entradas al mismo tiempo, pues la idea es ver que la aplicación sea capaz de manejar e indicarle al usuario que el valor de entrada es incorrecto en cada campo por separado. Si se ingresan valores de clases inválidas en ambos campos a la vez, tal vez no se logra verificar que para cada entrada incorrecta se hace un manejo adecuado.



The image shows a dialog box with a light gray background. Inside, there are two text input fields. The first field is labeled 'First Number' and contains the text '-100'. The second field is labeled 'Second Number' and contains the text '-4'. Below these fields, the text 'Error: Invalid Number' is displayed in a bold, black font. At the bottom of the dialog box, there are two buttons: a black button with the text 'Confirm' in white, and a white button with a black border and the text 'Cancel' in black.

Pantalla de ejemplo para la Técnica de No Enmascaramiento de Errores

En el ejemplo de la figura, imaginen que hay un error al ingresar un valor negativo en el campo correspondiente a *Second Number*, ya que no se controla correctamente que sea mayor que cero. Entonces, al probar de esta forma estamos “ocultando el error” o “enmascarando el error”.

Una buena forma para utilizar ambas estrategias de combinación de datos que nombramos sería: combinar primero a mano los inválidos según estas

consideraciones y luego, con las clases válidas, aplicar combinación por pares como vimos antes.

Calcular valores esperados

Para cada prueba diseñada, se calculan los valores esperados de acuerdo con las especificaciones del sistema o a nuestro conocimiento del negocio. En otras palabras, debemos diseñar el oráculo de prueba.

Esta es la única forma de determinar si la aplicación funciona como es esperado o no. En muchos casos, se vuelve fundamental que al momento de ejecutar las pruebas no se tenga que analizar los resultados, sino que sea posible comparar directamente con lo que ya estaba previsto. Es más, para el caso en que un experto del dominio desee diseñar las pruebas y que alguien más luego sea el encargado de ejecutarlas, dejar esto definido es de vital importancia, porque quizá quien ejecuta las pruebas no tiene total conocimiento de las reglas de negocio, entonces es deseable (conveniente) especificar los valores esperados de antemano.

Con *valores esperados* estamos incluyendo lo que el sistema muestra en pantalla, archivos que se tengan que modificar, estados de la base de datos o cualquier otro tipo de acción que deba realizar el *software* que estamos probando.

Así mismo, una buena práctica es incluir pruebas que deban registrar un fallo, para las que el resultado esperado sea el manejo de una excepción y su correcto procesamiento. Esto también es conocido como ***testing negativo***, que consiste en incluir escenarios con aquellas cosas que el sistema debe estar preparado para **no** hacer, o no fallar, por tratarse de una situación incorrecta. Por ejemplo, verificar que, si se intenta hacer una transferencia desde una cuenta bancaria sin saldo suficiente, la operación da error y no se restó el monto de la cuenta origen ni se sumó de la cuenta destino. Esto generalmente queda cubierto al diseñar pruebas con clases inválidas como vimos antes, pero no siempre es así, como por ejemplo, ver qué pasa si no existe el archivo de configuración, la tabla de clientes está vacía o qué ocurre si el sistema se queda sin conexión en la mitad del proceso.

Generalmente, se agrega alguna columna a la tabla de casos de prueba, donde se indican las verificaciones a realizar. Podrían ponerse dos columnas, por ejemplo, una para *valores de variables esperados* y otra para *acciones esperadas*. En la segunda columna se podría incluir cosas como “se despliega mensaje o se agrega registro a la tabla con los datos ingresados, se envía un *email*”.

Combinación de datos de prueba con casos abstractos o concretos

Veamos otro aspecto muy interesante, que es la posibilidad de diseñar las pruebas en forma abstracta, sin datos específicos o con los datos definidos en forma precisa desde su diseño. O sea, cuando estamos pensando los datos de prueba que vamos a utilizar para cada caso, podemos hacer la combinación a nivel de caso de prueba abstracto o a nivel de caso de prueba concreto con datos.

En el primer caso, los valores a combinar son del tipo:

- Números mayores que cero o números menores o igual a cero.
- Número válido, número inválido.
- Cantidad suficiente, cantidad insuficiente.
- Cliente existente, cliente que no existe.
- Número de cuenta con balance positivo, número de cuenta bloqueado, etc.

Como podrán ver, no estamos refiriéndonos a valores concretos, sino a características de los datos que vamos a utilizar. O sea, estamos indicando a qué clase de equivalencia deben corresponder. Entonces, como resultado vamos a tener casos de prueba que se podrán instanciar con distintos valores, siempre y cuando cumplan esas restricciones marcadas.

La otra opción es elegir directamente representantes de las clases de equivalencia y combinar esos representantes. Por ejemplo, haciendo referencia al ejemplo anterior:

- 45, -3
- 4, "4%,daa"

- 1000, 1
- Juan Pérez, ClienteInexistente Rodriguez
- 11223344, 232323, 445500

En este caso, los datos a los que se hace referencia (como los clientes o los números de cuenta) tienen que existir en la base de datos y debemos saber que cuentan con esas características deseadas. En el caso de *Cantidad suficiente*, se refiere generalmente a la relación entre dos variables, como en el caso de crear una factura, la relación entre la cantidad de un producto que se desea facturar y la cantidad existente en *stock*. En ese caso, será necesario también controlar estas cantidades para asegurarse de que la condición se cumpla, o sea, que se está probando la situación deseada, las clases de equivalencia seleccionadas.

Claramente, la primera opción es más legible, más orientada al *tester* que está leyendo una planilla de pruebas y va ejecutando contra el sistema; necesita saber qué está ejecutando para conocer bien cuál es el comportamiento esperado. El segundo caso, tal vez sea el más adecuado para el *testing* automático, en el que es necesario indicar exactamente qué valores hay que utilizar para la ejecución, ya que no habrá una mente humana en el medio.

Combinación de datos de prueba relacionados o dependientes

Veamos un ejemplo un poco más complejo, pero muy típico en cualquier sistema que vayamos a probar.

El mecanismo explicado hasta aquí para combinar datos tiene una precondition importante: las variables a combinar deben ser independientes. Muchas veces necesitamos también combinar datos de variables dependientes, que no cumplen esa precondition.

Imaginemos que estamos probando el ingreso de facturas, para lo cual es necesario indicar el cliente al que se factura. Asociado a la factura, tenemos variables como la fecha, el tipo de pago, los productos que se facturan (indicando la cantidad de cada uno), etc. y a su vez, asociado a cada producto tenemos el precio y el *stock*

disponible. La creación de facturas tiene distintos escenarios que dependen de si el cliente es moroso o no, o si es mayor de edad o no. Entonces, nos va a interesar combinar estas variables con el resto de las variables que están involucradas (por ejemplo, los distintos tipos de pago deben ser probados con clientes morosos y no morosos). La morosidad del cliente, la edad y, quizá, otros atributos están relacionados con el cliente seleccionado, así que son variables dependientes y no las podemos combinar de forma tan directa.

En estos casos proponemos lo siguiente:

En primera instancia definimos las clases de equivalencia para los clientes, por ejemplo:

- Cliente moroso.
- Cliente no moroso.
- Cliente mayor de edad.
- Cliente menor de edad.

O incluso, pensamos las clases de equivalencia de cada variable y luego las combinamos. De esta forma, vamos a tener los distintos clientes interesantes a utilizar. Siguiendo el ejemplo, podríamos tener un cliente menor de edad, un cliente mayor de edad que es moroso y un cliente mayor de edad no moroso (estamos considerando una regla de negocio que indica que los menores de edad no pueden tener mora, con lo cual eliminamos esa combinación del conjunto de pruebas).

Una vez que los definimos, los creamos en nuestro sistema y así obtendremos su identificador (ya sea un ID numérico o el nombre del cliente, dependiendo de la implementación). Esto sería como crear representantes de nuestras clases de equivalencia.

Luego, cuando estamos pensando datos de prueba para crear facturas, hacemos la combinación de valores tomando para los clientes su nombre o identificador y sin combinar las variables dependientes (mora y edad), pues estas ya están combinadas en los distintos clientes que estamos usando.

Datos para la creación de Factura:

- Cliente.
- Tipo de pago.
- Fecha.
- Producto.
- Etc.

Esto aplica a cualquier instancia o cualquier variable dependiente. Incluso, para este ejemplo, también deberíamos hacer algo similar para los productos, ya que, claramente, el producto seleccionado, el precio y el *stock* no son independientes, sino que el precio y el *stock* se derivan del producto elegido.

El caso del producto es un poco más complejo aún, porque también se debe indicar la cantidad facturada de cada uno y será interesante jugar con distintos valores considerando una relación entre el *stock* y la cantidad (facturar más de la cantidad en *stock*, menos, exactamente lo mismo, etc.).

Comentarios finales del capítulo

Hasta acá fue la base, la introducción al *testing*, lo más elemental para diseñar y ejecutar pruebas. Quizá algo más para comentar es que, lamentablemente, hay muchas personas que se desempeñan como *testers* en la industria, quienes ni siquiera manejan estas técnicas básicas, entonces, si ustedes se afianzan con lo propuesto en este capítulo, ya van a estar en condiciones para ser *testers* bastante competentes.

Lo que viene a continuación hace que elevemos el nivel del juego mucho más, lo cual implica aplicar técnicas más avanzadas para modelar y diseñar pruebas, usar heurísticas, seguir enfoques de *Testing Exploratorio*, y luego ver cómo incluso agregar automatización a nuestro *testing*, tanto para la verificación funcional como para análisis de rendimiento (*performance*).

Lo mejor es enemigo de lo bueno.

Voltaire

TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA

Existen distintas técnicas, y cada vez que estemos diseñando pruebas veremos que hay algunas que son más apropiadas que otras, según el caso. Al aprender a usarlas, veremos en qué tipo de situaciones nos dan más valor. Mientras que las técnicas básicas se utilizan siempre, las que veremos ahora las utilizaremos solo cuando queramos lograr una mejor cobertura o sean específicas para la situación que estemos probando. Por ejemplo, si el sistema o funcionalidad bajo pruebas se caracteriza por contar con mucha lógica basada en condiciones, nos convendrá aplicar la técnica de **tablas de decisión** o la de **grafo causa-efecto**. En el caso de que la aplicación varíe su comportamiento de acuerdo a distintos estados que se puedan definir en ella, entonces, será aplicable la de **máquinas de estado**. Hay una técnica para desarrollar pruebas a partir de **casos de uso** (aunque incluso sin casos de uso documentados, podemos comenzar por construir esa documentación a través de entrevistas a las personas expertas del dominio, explorando la aplicación o viendo la documentación que haya disponible). Si estamos probando la creación, edición, borrado y lectura de las entidades del sistema, entonces, podremos aplicar la técnica llamada **Matriz CRUD**. En esta sección, veremos estas técnicas, utilizando ejemplos y viendo cómo aplicarlas paso a paso.

Derivación de casos de prueba desde casos de uso

Como en la mayoría de las técnicas no hay una única forma de aplicar, en la que veremos a continuación, para entender cómo funciona luego de describir cómo se modelan los casos de uso, explicaremos cómo se aplica, siguiendo un ejemplo. En ese caso, veremos que habrá que tomar muchas decisiones. Vamos a plantear algunas alternativas o cuestiones que podríamos tener que considerar, pero queremos remarcar el hecho de que no hay que dejar de pensar, ¡nunca!

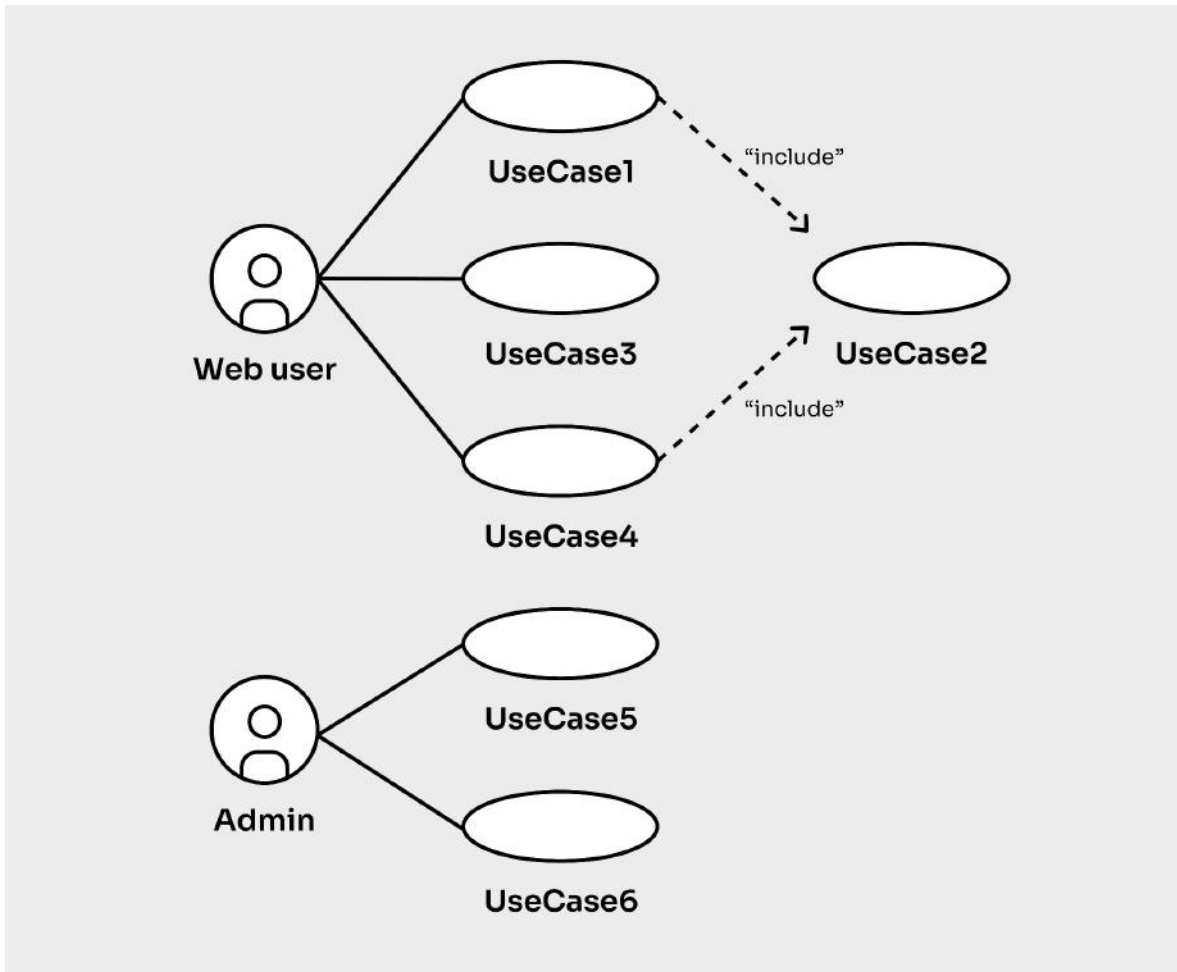
Representación de casos de uso

Los **casos de uso** son un elemento de análisis muy conocido para documentar una interacción típica entre el usuario y el sistema (**no confundir con caso de prueba**). Ganaron su popularidad incluso con UML²² y los diagramas de casos de uso, como el que aparece en la siguiente figura. Ahora, estos diagramas UML simplemente muestran los actores involucrados en cada caso de uso (muestran qué tipo de actor o usuario puede ejecutar cada caso de uso o funcionalidad) y la relación entre un caso de uso y otro (relaciones de dependencia o inclusión). En esta técnica, no utilizaremos ese tipo de diagramas, sino las descripciones de los casos de uso, lo que nos indica cuál es la interacción esperada entre el usuario y el sistema para realizar determinada acción o lograr determinado cometido.

Con esta técnica de modelado, se captura conocimiento del negocio y de cómo se desea que trabaje el sistema, se describe lo que debe hacer sin entrar en detalles. Por este motivo –de estar disponible– es una fuente interesantísima para utilizar como *input* para diseñar pruebas. Analizando cada caso de uso, podemos generar las pruebas que cubran los distintos escenarios de los requisitos del sistema que estamos probando.

²² UML (Unified Modeling Language): <<https://www.uml.org>>

Lo que ocurre muchas veces es que no se cuenta con las formalizaciones de los casos de uso. Estos quizá están en la mente de distintas personas que ocupan diversos roles (analistas, desarrolladores, usuarios, etc.). El equipo de *testing* muchas veces termina aportando en la formalización de la documentación, para poder luego validar los documentos y utilizarlos como el *input* principal para el diseño del oráculo de pruebas.



Ejemplo de diagrama UML de casos de uso

Veamos las distintas formas de representar el caso de uso.

Representación tabular

Los casos de uso se representan en lenguaje natural, pero generalmente se intenta seguir cierta estructura, indicando el **flujo de interacción** como pasos numerados. Habitualmente, se comienza describiendo el **flujo principal** y luego se enriquece describiendo cada uno de los posibles **flujos alternativos** y **excepciones**.

Cada caso de uso cuenta con una serie de **precondiciones** y **postcondiciones** que deben cumplirse antes y luego de su ejecución. Se suele definir también un objetivo o **descripción** asociado al caso de uso, indicando qué es lo que buscará el usuario al ejecutar ese caso de uso. Al ver la siguiente tabla, podemos observar un ejemplo simple de cómo se podría describir un caso de uso para el acceso a un sistema, considerando que es necesario contar con una cuenta asociada a un *email* y *password*.

Nombre	Acceso al sistema
Autor	Federico Toledo
Fecha	09 / 01 / 2014
Descripción Un usuario debe registrarse para hacer uso del sistema y para ello debe hacer <i>login</i> con su <i>email</i> y <i>password</i> . Si no tiene una cuenta, debe registrarse en el sistema creando así su cuenta.	
Actores Usuario a través de la interfaz <i>web</i> .	
Precondiciones El usuario debe contar con una cuenta de <i>email</i> válida.	
Flujo normal <ol style="list-style-type: none">1. El usuario accede al sistema en la URL principal.2. El sistema solicita credenciales.3. El usuario ingresa <i>email</i> y <i>password</i>.4. El sistema valida las credenciales del usuario y le da la bienvenida.	

<p>Flujo alternativo 1</p> <ol style="list-style-type: none"> 3. El usuario no recuerda su <i>password</i>, por lo que solicita que se le envíe por <i>email</i>. 4. El sistema solicita el <i>email</i> y envía una nueva clave temporal al <i>email</i>. 5. Repite pasos 3 y 4 del flujo normal, pero con clave temporal enviada por <i>email</i>. <p>Flujo alternativo 2</p> <ol style="list-style-type: none"> 3. El usuario no está registrado en el sistema, por lo que solicita crear una cuenta. 4. El sistema solicita los datos necesarios para crear la cuenta. 5. El usuario ingresa los datos y confirma. 6. El sistema crea la cuenta del usuario. 7. Con la cuenta ya creada, se inicia el flujo normal.
<p>Excepciones</p> <ol style="list-style-type: none"> E1. (Flujo normal) <i>Email</i> y/o <i>password</i> incorrectos. Si esto sucede tres veces consecutivas, la cuenta del usuario se bloquea por seguridad. E2. (Alternativo 1). El <i>email</i> proporcionado no está registrado en el sistema. El sistema notifica el error.
<p>Postcondiciones</p> <p>El usuario accede al sistema y se registra su acceso en la tabla de registro de actividad.</p>

Ejemplo de un caso de uso simple

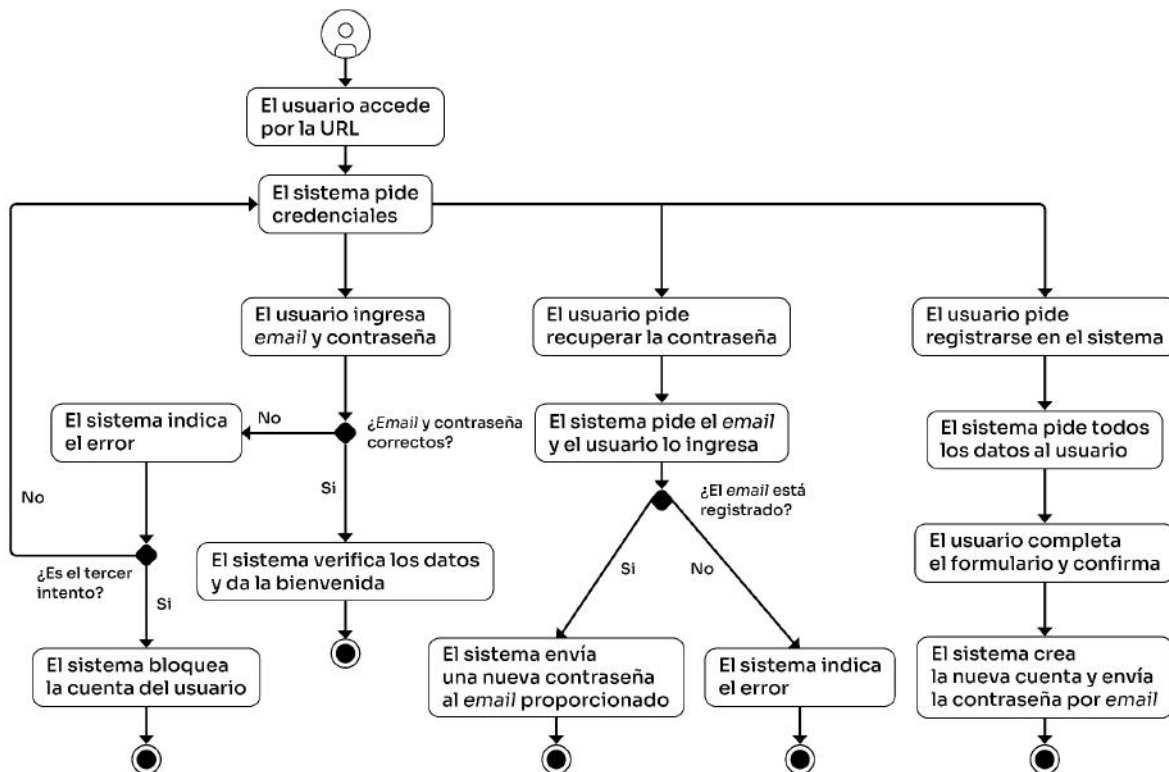
Representación gráfica

Para aplicar la técnica de generación de casos de prueba a partir de casos de uso, lo primero será pasar a otra representación, en forma de grafo. Con esta representación, se podrá visualizar más fácilmente los distintos flujos que se pueden seguir y seleccionar como casos de prueba.

Nosotros les vamos a sugerir aquí una representación intermedia, que es mucho más completa y permitirá validar ideas con mayor detalle. De cualquier forma, este paso podría saltarse en caso que el flujo ya esté claro para todos los integrantes de equipo y no genere valor agregado.

El beneficio principal al pasar a una representación gráfica es abstraerse de la letra, representarlo como un grafo dirigido que muestre fácilmente cuáles son los flujos del sistema. Esta técnica es extrapolable casi que para cualquier especificación del sistema que podamos trasladar, no exclusivamente a casos de uso representados en el formato tabular mostrado anteriormente.

Vamos a utilizar para esto un diagrama de actividad, donde quedarán representados los distintos flujos del caso de uso. Al observar la siguiente figura, se puede ver que, al representar el caso de uso como un diagrama de actividad, todos los flujos pueden ser representados en forma concisa y unificada. Además, cuando uno trabaja en diseñar el modelo, necesita analizar y refinar el entendimiento de detalles que pueden ser muy útiles para el *testing*.



Ejemplo de diagrama de actividad para caso de uso de *login*

Si tienen alma de *testers*, seguro que están buscando errores en la documentación y representación, ¿no? Les planteamos como **ejercicio** analizar las dos representaciones y ver así qué información se agregó al pasar al modelo. Al aplicar esta técnica en un caso real, surgen muchas dudas que deben ser consultadas con algún experto sobre la aplicación y el negocio y, quizá, aquí ya surjan oportunidades de mejora con solo analizar el modelo. A modo de ejemplo, donde dice *El sistema pide todos los datos al usuario*, ¿cuáles son todos los datos? En el caso de uso escrito indicaba *los datos necesarios*. En cualquier caso, no se están

especificando, y parte de nuestro trabajo es identificar esas ambigüedades en la documentación, ya que pueden dar origen a problemas o expectativas no colmadas.

Con una representación de este estilo, y según con el tiempo con el que se cuente para probar, podremos decidir qué flujos queremos probar. Con este tipo de representación, suele ser mucho más fácil de visualizar y analizar que la representación textual.

Derivando los casos de prueba

Básicamente, para derivar los casos de prueba vamos a **recorrer desde el nodo inicial a cada uno de los nodos finales**, pasando por cada una de las transiciones del modelo. En el caso que existan bucles, se deberá decidir si es suficiente con visitar una sola vez el bucle, o si vale la pena recorrerlo varias veces y, en ese caso, cuántas. Una vez que tengamos los flujos vamos a analizar qué datos utilizar para cada uno.

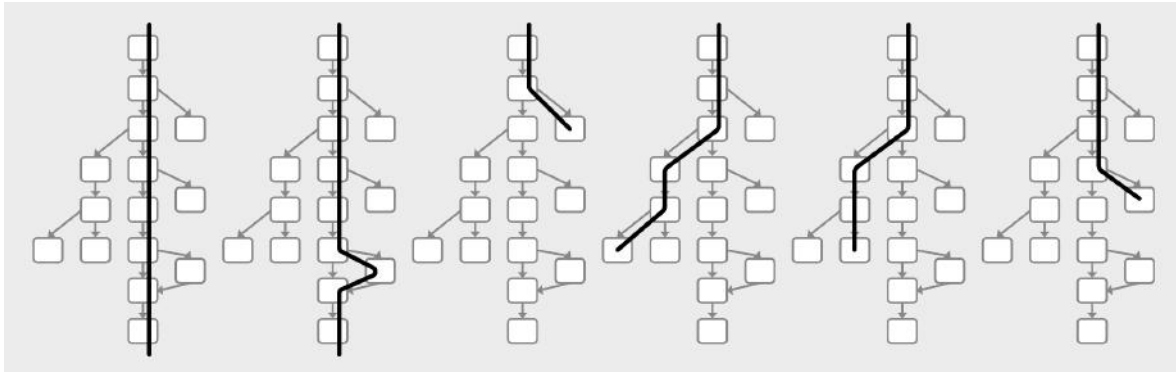
Flujos de prueba

Podríamos seleccionar distintos criterios de cobertura sobre este diagrama (todas las transiciones, todos los nodos, todas las decisiones, todos los escenarios, combinación por pares de entrada/salida de cada nodo, etc.), pero por ahora simplificaremos al respecto y consideraremos solo un criterio de cobertura. Se analizarán más opciones al respecto cuando veamos la técnica de máquinas de estado, las cuales son directamente aplicables para estos diagramas.

Para seleccionar los flujos, debemos tener en cuenta dos cosas, principalmente:

Flujos alternativos: cada uno de los flujos alternativos debería ser recorrido al menos una vez. Con esto, nos garantizamos visitar cada posible interacción del usuario con el sistema, y cada grupo de datos que hace que se vaya por un flujo u otro. Para ejemplificar, la siguiente figura muestra cómo se podrían seleccionar los flujos en un grafo correspondiente a un diagrama de actividad con distintas alternativas (en este ejemplo, no se incluyen bucles). Este grafo muestra el flujo

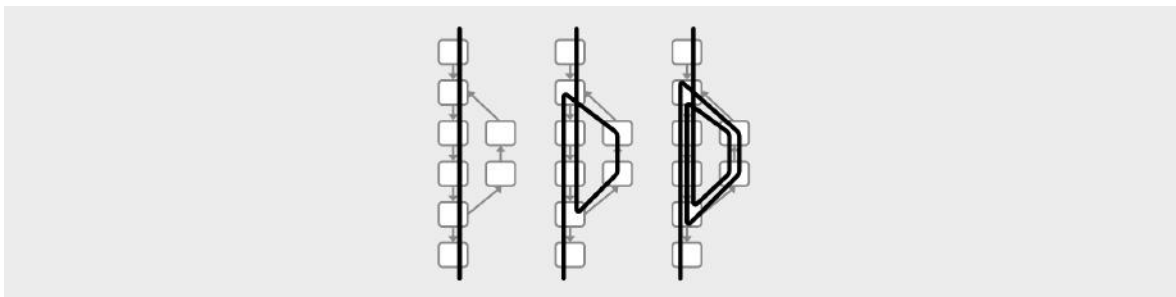
principal en la vertical y los distintos flujos alternativos saliendo de él. En negro están marcados los flujos que se derivarían como casos de prueba.



Flujos derivados del grafo del caso de uso

Bucles: para estos casos deberíamos al menos seleccionar un caso en el que no se ejecute el bucle, uno en el que se ejecute una vez y, en algunos casos, si se considera que es oportuno, repetir el bucle varias veces. Se podrían distinguir dos tipos de bucles, los que tienen una cantidad de repeticiones definida (como en el ejemplo, después de tres repeticiones cambia el flujo) o las que no tienen una cantidad de repeticiones definida. Generalmente, esto último se da cuando se trata de agregar ítems a una lista o similar.

El ejemplo de la siguiente figura muestra que para un caso en el que hay un ciclo se pueden definir tres casos de prueba: uno que no ejecuta el bucle, uno que lo ejecuta una vez y uno que lo ejecuta tres veces.



Flujos derivados del grafo del caso de uso con ciclos

En el ejemplo que veníamos trabajando, existe un bucle para cuando el usuario ingresa el *email* o contraseña incorrectamente. En este caso hay una situación

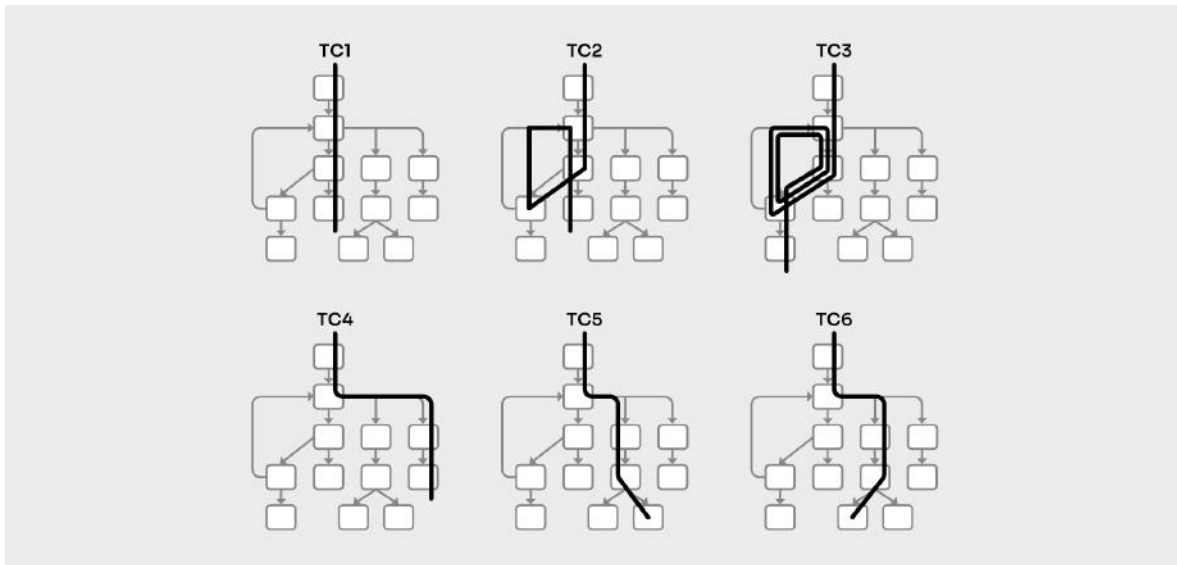
especial, que es cuando este bucle se ejecuta tres veces consecutivas. En este caso es claro que vamos a querer ejecutar el ciclo: una vez sin entrar en él, una vez y por último tres veces, para que se produzca esa situación y finalice en el camino alternativo en el que se bloquea la cuenta de usuario. De hecho, es necesario hacer esto si queremos cubrir todas las transiciones.

Entonces, para cubrir todas las transiciones del ejemplo anterior, considerando las bifurcaciones y los bucles tal como acabamos de explicar, deberíamos considerar al menos los seis casos de prueba resumidos en la siguiente tabla.

Caso de prueba	Descripción
TC1	Flujo principal. El usuario ya registrado accede con el <i>email</i> y contraseña correctos.
TC2	El usuario se equivoca al ingresar <i>email</i> y contraseña, pero en su segundo intento lo hace bien y accede al sistema.
TC3	Como el usuario falla en tres intentos ingresando <i>email</i> y contraseña, el sistema no lo deja ingresar y bloquea su cuenta.
TC4	El usuario es nuevo en el sistema, por lo que solicita registrarse. El sistema crea una cuenta, le asigna una contraseña y se la envía por <i>email</i> .
TC5	El usuario ya está registrado, pero no recuerda su contraseña, por lo que le pide al sistema que le envíe una nueva. El usuario ingresa una dirección de <i>email</i> que no corresponde a ningún usuario registrado, por lo que se termina indicando el error.
TC6	El usuario ya está registrado, pero no recuerda su contraseña, por lo que le pide al sistema que le envíe una nueva. El sistema genera una y se la envía al <i>email</i> .

Casos de prueba para el ejemplo del *login*

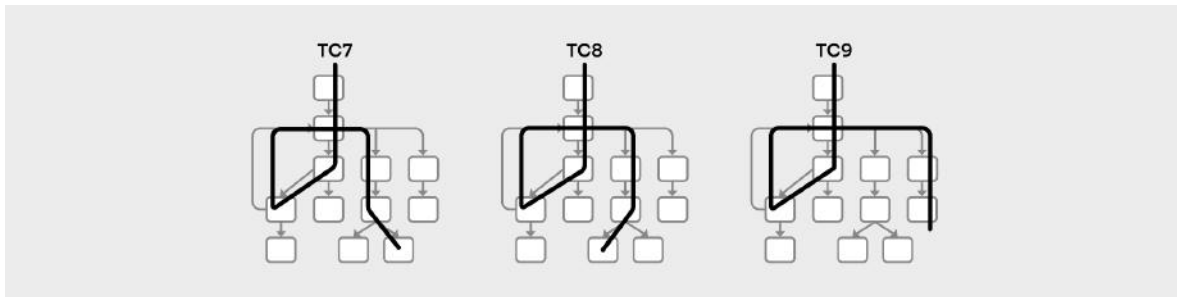
En la siguiente figura se observa a qué flujo corresponde cada caso de prueba, y así se puede observar cómo se obtuvieron los casos de prueba a partir del diagrama.



Flujos de los casos de prueba de *login*

Observen que, como el caso de uso tiene un ciclo, en los casos de prueba generados luego de ejecutar una repetición en TC2 seguimos por el flujo principal y en el caso del TC4, TC5 y TC6 no se consideraron entradas al ciclo. En este ejemplo, consideramos que no era interesante probar qué pasa en esos flujos luego de cometer algún intento fallido al intentar introducir *email* y contraseña . Habrá casos en los que sea interesante considerar las combinaciones de las repeticiones al ciclo con los distintos flujos que se pueda seguir luego. Si así fuera el caso, deberíamos agregar tres casos más que se muestran en la siguiente figura.

Tenemos un total de nueve casos. Esto se explica (o se pudo haber calculado de antemano) viendo que tenemos cuatro caminos para seguir sin el flujo, luego podemos decidir entrar o no entrar y esto multiplica los cuatro caminos por dos, y por último debemos sumar uno que corresponde al flujo alternativo de ejecutar tres repeticiones del ciclo.



Casos de prueba combinando el ciclo

Esto sucede también cuando se pueden combinar varios ciclos.

Datos de prueba

Luego de diseñar los flujos, habrá que decidir **qué datos usar** en cada uno. Para esto combinaremos con otras técnicas como las ya vistas en secciones anteriores, por ejemplo, usando partición en clases de equivalencia, valores límites, combinación por pares, etc., una vez que hayamos reconocido las variables en juego.

Aquí hay algo particular a considerar: **ciertos flujos se asocian con ciertas clases de equivalencia de algunas variables**. O sea, solo podremos seleccionar algunas clases de equivalencia para algunos flujos. Por ejemplo, el flujo del TC5 (donde se quiere recordar la contraseña y para ello se debe ingresar la cuenta de *email* registrada) solo se puede ejecutar si se selecciona una cuenta de *email* válida, con lo cual para ese caso de prueba no consideraremos todos los valores posibles para esa variable, sino que solo los que aplican para el flujo correspondiente. De hecho, hay algunas variables que ni siquiera están en juego, como la contraseña. El flujo del caso de prueba indica que la contraseña no es una entrada que cambie el comportamiento de ese caso, con lo cual no es variable para este caso, por lo tanto, no tiene sentido diseñar valores de prueba para la variable *contraseña* para ese caso en particular.

Lo que se suele hacer en este punto es definir cuáles son las variables para cada flujo y cuáles son las clases de equivalencia a considerar. De esa forma, luego vamos a poder determinar valores de prueba para ellos.

En la siguiente tabla se muestran las variables y sus categorías para cada caso de prueba derivado en el ejemplo. Para simplificar, solo se consideraron dos particiones de equivalencia para cada variable: *válida* e *inválida*.

Se puede ver fácilmente que, si bien hay muchas variables en juego y cada una puede tener diversas clases de equivalencia identificadas, no todas son combinables. Las variables que tenga cada caso de prueba dependerán de su flujo.

Variables =>	<i>Email y contraseña</i>	<i>Email para recordar contraseña</i>	<i>Datos para crear cuenta (nombre, email, contraseña)</i>
Caso de prueba			
TC1	1 par válido		
TC2	1 par inválido 1 par válido		
TC3	2 pares inválidos 1 par válido		
TC4			1 conjunto de datos
TC5		1 válido	
TC6		1 inválido	
TC7	1 par inválido	1 válido	
TC8	1 par inválido	1 inválido	
TC9	1 par inválido		1 conjunto de datos

Datos de prueba para el ejemplo

A su vez, con *email* y contraseña se puede jugar con los valores *par válido* y *par inválido*. Es decir, probar un *email* válido y una contraseña inválida, un *email* inválido (y en ese caso la contraseña no importa), etc. Lo que importa en este caso es si el par es válido o no. En el caso del *email*, será necesario probar el TC6 (el caso en que el usuario ya está registrado, pero no recuerda su contraseña) con un *email* inválido, lo cual puede incluir desde *emails* no registrados hasta *emails* con formato incorrecto. En el caso de los TC4 y TC9 tenemos distintas combinaciones de datos, porque si bien no se muestran en la tabla, hay un conjunto de variables asociadas a la creación de la cuenta de usuario. Si bien tampoco se indica qué pasa si el usuario ingresa información incorrecta, igualmente deberá ser tenido en cuenta para ese paso.

Actores

Otro aspecto que debería ser tenido en cuenta es qué actores pueden ejecutar el caso de prueba. Este tipo de información generalmente está disponible en las descripciones de los diagramas de uso e, incluso, en los diagramas UML de casos de uso, muestra qué actores están relacionados con cada caso.

Si siempre probamos con el usuario *admin*, no vamos a tener ningún problema de permisos, por ejemplo, pero ¿qué pasará luego con el usuario *web* que tiene acceso restringido a algunas partes del sistema? No proponemos duplicar los casos de prueba por cada actor (o tipo de usuario) que pueda ejecutar el caso de uso, pero sería deseable en al menos algún caso, quizá el flujo principal, que se ejecute con distintos actores con distintos permisos.

Selección de casos de prueba

Hasta acá vimos la técnica más o menos completa. Eso no significa que vamos a querer ejecutar todos los casos de prueba con todas sus combinaciones de datos y para cada caso de uso del sistema. Esta es la técnica que nos da la cobertura de casos de uso, luego está en nosotros seleccionar lo que ejecutamos de acuerdo a los recursos y la importancia que le damos a cada valor de cada variable y a cada caso de uso.

Esto también nos puede ayudar a establecer una medida de la calidad de las pruebas que podemos ejecutar. Imaginen que con esta técnica generamos un total de 100 casos de prueba (contando todos los flujos y los datos que se puedan usar), pero solo podemos ejecutar 75. *OK*, entonces vamos a poder dar más información: siguiendo la definición tradicional de cobertura (escenarios seleccionados/escenarios totales) tenemos una cobertura del 75% según la técnica de casos de uso.

¿Cómo seleccionar? Pues, teniendo en cuenta lo que antes dijimos y, de esa forma, priorizando:

1. Primero, seleccionando los casos de uso más importantes.
2. Para cada caso de uso, cuáles son los flujos más importantes.

3. Para cada flujo, ver qué datos son los más importantes a utilizar.

Cuando decimos *importante*, hablamos de *testing* basado en riesgos, considerando relevancia, lo más usado, lo que más costos o impacto negativo genera en el caso de no funcionar, lo que está más “verde” (o menos probado y seguramente con más probabilidades de que tenga errores), etc.

Técnica de Tablas de Decisión

La técnica llamada Tablas de Decisión es muy aplicable cuando la lógica a probar está basada en decisiones, principalmente, si se puede expresar la lógica en forma de reglas tales como:

- *Si A es mayor que X entonces...*
- *Si el cliente C tiene deuda entonces...*
- etc.

O, dicho de otra forma, aplica a programas donde la lógica predominante es del tipo *if-then-else*. También tiene sentido aplicarla cuando existen relaciones lógicas entre variables de entrada o ciertos cálculos que involucran subconjuntos de las variables de entrada. En estas situaciones se procura encontrar errores lógicos o de cálculo, principalmente en las combinaciones de distintas condiciones.

Para trabajar con estas situaciones y abstraernos de una descripción en lenguaje natural de las distintas reglas que rigen al sistema que estemos probando, vamos a buscar una representación utilizando una estructura lógica.

Veremos dos técnicas aplicables para estas situaciones: primero, en esta sección, veremos la representación con **Tablas de Decisión** para poder derivar los casos de prueba con ella y segundo, en la siguiente sección, veremos una forma de representación llamada **Grafo Causa-Efecto**. Esta última, como podrán imaginar, implica tener un modelo para representar las causas y los efectos que están relacionados, indicando criterios lógicos.

La tabla de decisiones la vamos a poder completar con las combinaciones de datos que formarán los **casos de prueba**. Básicamente, se listan las condiciones y acciones identificadas en una matriz. Luego, debemos identificar cuáles son los posibles valores para esas condiciones y cómo están relacionadas. Estas, las llenaremos en la matriz en una manera especial, de forma que al terminar de completarla tendremos las combinaciones que conformarán los casos de prueba.

Cada columna indicará una posible combinación entre condición y acción. De esta forma, las modelamos en una estructura lógica.

Ejemplo bajo pruebas

Veamos un **ejemplo** para mostrar cómo representar un conjunto de reglas, considerando que estamos probando un sistema para la determinación de tratamientos, basándose en el índice de masa corporal (IMC), la edad y el sexo.

Primero, veamos cuáles son los valores que forman parte de las condiciones o causas en las distintas variables que están en juego (masa corporal, edad y sexo).

El sistema maneja el índice de masa corporal, que puede estar en distintos rangos:

- $A = \text{IMC} < 18,5$.
- $B = 18.5 < \text{IMC} < 30$.
- $C = \text{IMC} > 30$.

Se distinguen 4 franjas de edad:

- B = Bebé, menos de 2 años.
- M = Menor, de 2 a 18.
- A = Adulto, de 18 a 50.
- AM = Adulto mayor, mayor de 50.

Las **reglas de negocio** consideradas para el ejemplo son las siguientes:

- Para los hombres, de acuerdo a la edad se les asigna una dosis distinta. Para las mujeres, la edad es indistinta.
- De acuerdo al IMC, se tendrá en cuenta la dosis de una alimentación suplementaria. Las mujeres necesitarán un complemento M. Si se trata de hombres, a los bebés deberá suministrarse el complemento X, si son menores el complemento Y y, si son adultos o adultos mayores, el complemento Z.
- En el caso de hombres adultos mayores, deberá realizarse un análisis extra en el caso que su IMC esté en la franja A.

Derivando los casos de prueba con Tablas de Decisión

Gracias a la técnica, se pueden definir combinaciones de los datos de prueba, considerando las distintas condiciones, sin generar datos inconsistentes ni redundantes. Los pasos a seguir son los que se muestran a continuación:

1. **Listar todas las variables** y las distintas condiciones a incluir en la tabla de decisión: en el ejemplo podríamos decir que son los distintos valores que pueden tomar las variables sexo, IMC y edad.
2. **Calcular la cantidad de combinaciones posibles** y con eso definimos cuántas columnas debemos designar.
3. **Agregar las acciones a la tabla.** Las reglas de negocio indican qué acciones se disparan para cada combinación de datos de entrada, con lo cual para cada posible combinación deberemos analizar las reglas manualmente y determinar la salida esperada.
4. **Verificar la cobertura de las combinaciones**, asegurando que con las combinaciones que vamos a seleccionar estaremos cubriendo el total del espacio de valores.

Hasta aquí, listamos las condiciones y sus posibles valores. Esto nos permite comenzar a armar la tabla colocando las condiciones en la columna izquierda. De esta forma, comenzaremos diseñando la estructura tal como se ve en la siguiente tabla. En esta se incluyeron tres filas para definir las condiciones y, luego, tendremos distintas filas para incluir las acciones asociadas. En cada nueva columna se completará una combinación de datos que corresponde a un nuevo caso de prueba.

Test Cases => Condiciones	1	2	3	4	5	6	...
IMC							
Sexo							
Edad							
Acciones							

Estructura de la tabla de decisión

A continuación, calcularemos cuántas combinaciones tendremos para este ejemplo. Esto es simple de calcular, multiplicando las cantidades de opciones para cada condición; serán entonces $2 \times 3 \times 4 = 24$ combinaciones posibles. En la siguiente tabla se reflejan las combinaciones interesantes (las más relevantes, las más usadas, las que generan mayor impacto negativo en caso de no funcionar, etc.) de acuerdo a las acciones que toma el sistema.

Para poder armar la tabla, debemos tener en cuenta algo llamado **factor de repetición** para cada variable. Básicamente, es necesario analizar las condiciones involucradas y multiplicar la cantidad de valores de cada una. Por ejemplo:

- En la condición *Sexo*, el valor *Mujer* no está involucrado con otras condiciones, el factor de repetición es 1 y, por lo tanto, este valor ocupará 1 columna de la tabla de decisión. En cambio, el valor *Hombre* está vinculado con la condición *IMC* (3 valores) y *Edad* (4 valores), por lo que necesitamos $3 \times 4 = 12$ columnas para ese valor (ver la fila *Sexo* de la tabla).
- En la variable *IMC* cada valor está relacionado con *Edad*, por lo que necesitamos 4 columnas para cada una.

La tabla de decisiones quedaría compuesta por las distintas condiciones y sus posibles valores combinados. El *tester* completa la tabla con las acciones que se esperan en el sistema ante cada combinación de datos (para cada columna).

Por último, se analiza columna por columna y se determinan las acciones que debería verificarse para cada caso de prueba. De esta forma, cada columna es un caso de prueba y las acciones son los resultados esperados, que se validarán. Es una forma ordenada de probar todas las condiciones importantes que maneja el sistema.

Condiciones				Combinaciones									
Test cases	1	2	3	4	5	6	7	8	9	10	11	12	13
Sexo	M	H	H	H	H	H	H	H	H	H	H	H	H
IMC	-	A	A	A	A	B	B	B	B	C	C	C	C
Edad	-	B	M	A	AM	B	M	A	AM	B	M	A	AM
Acciones	Dosis Mujer	Dosis H-B	Dosis H-M	Dosis H-A	Dosis H-AM	Dosis H-B	Dosis H-M	Dosis H-A	Dosis H-AM	Dosis H-B	Dosis H-M	Dosis H-A	Dosis H-AM
	Comp Mujer	Cto.X	Cto.Y	Cto.Z	Cto.Z	Cto.X	Cto.Y	Cto.Z	Cto.Z	Cto.X	Cto.Y	Cto.Z	Cto.Z
					Análisis								

Casos de prueba derivados

En la tabla, el guion (“-”) indica que cualquiera de las opciones puede ser utilizada en esa variable para el caso de prueba de esa columna. De esta forma, se redujo el número de combinaciones de 24 a 13, probando toda la lógica de las acciones presentadas para el ejemplo.

Cada columna representa distintas combinaciones, ya que cada “-” que se coloque en la tabla corresponderá a todas las condiciones posibles de esa decisión. De esta forma, se podría verificar que las combinaciones resultantes de esta tabla cubren a todo el producto cartesiano (todas las posibles combinaciones).

El caso de prueba 1 representa 12 combinaciones (ya que incluye cualquier combinación de *IMC* con *Edad*, lo que da $3 \times 4 = 12$). Si sumamos todas las combinaciones, nos da un total de 24, que como ya comentamos corresponde al total de combinaciones posibles. De esta forma, queda verificado que estamos cubriendo todas las combinaciones.

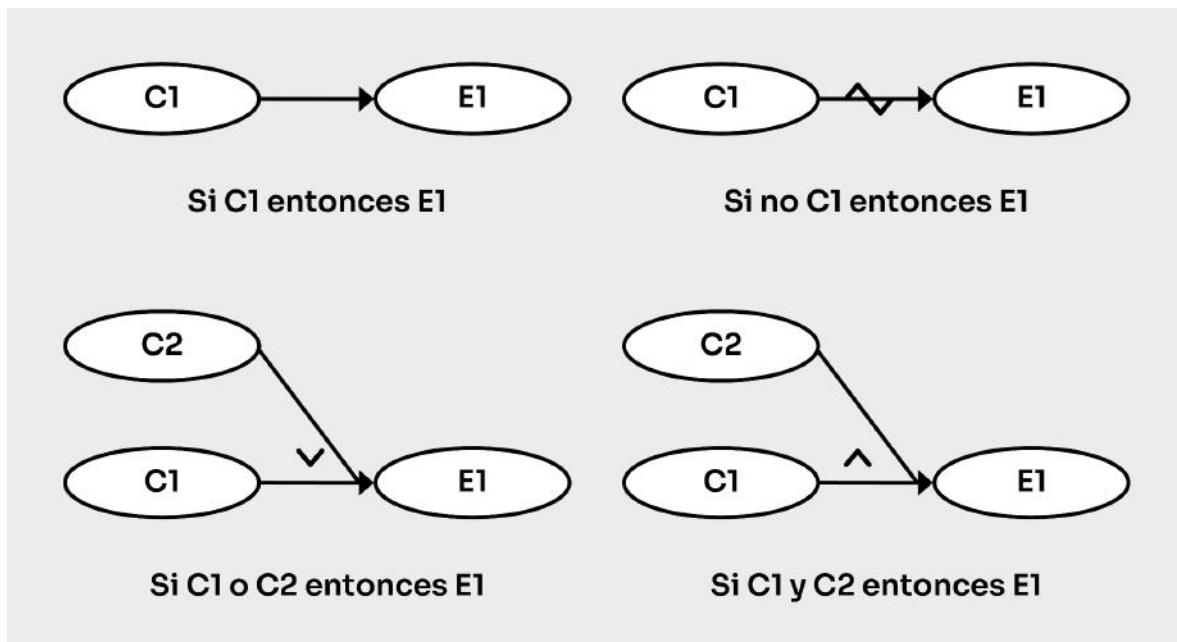
Es importante destacar que, si no armábamos la tabla en el orden adecuado, la cantidad de casos de prueba resultantes sería distinta y los casos de prueba resultantes serían redundantes. Por ejemplo, hay que analizar qué variable es más restrictiva que otra, pensando en cuál te ayuda a recortar combinaciones antes. En la tabla se ve que pusimos *sexo* arriba, lo cual permitió poner un guion abajo de la *M*. De no haberlo hecho así (dejando *sexo* para el final), estábamos obligados a tener que abrir cada caso y así se generan más columnas, más combinaciones y así más casos de prueba para cubrir la misma lógica.

Técnicas de Grafos

Causa-Efecto

Los grafos causa-efecto representan la relación lógica entre distintas causas y los posibles efectos. Para esto se listan las causas (entradas o acciones del usuario) y los efectos (salidas o acciones del sistema esperadas) y, luego, se unen indicando relaciones entre ellos.

Con esta representación, estaremos mostrando las reglas de la lógica del sistema. La siguiente figura muestra los constructores básicos para representar estos grafos (considerando los nodos C como causas y los nodos E como efectos), así como los operadores lógicos que se pueden utilizar para representar las reglas de negocio.



Representación con grafos causa-efecto

Vale aclarar que se pueden agregar nodos intermedios a modo de representar la lógica combinada de distintas opciones.

Luego, esta representación gráfica se pasa a una tabla que muestra las distintas combinaciones y reglas que se extraen de lo que ahí está modelado. Para eso, se ingresan primero todas las causas, o sea, una fila por cada nodo que está a la izquierda de una relación causa-efecto. Luego, se agrupan también los efectos, destinando una fila por cada nodo que aparece a la derecha en una relación causa-efecto. Las distintas columnas se llenan con las relaciones de las causas y efectos.

Causas						
Efectos						

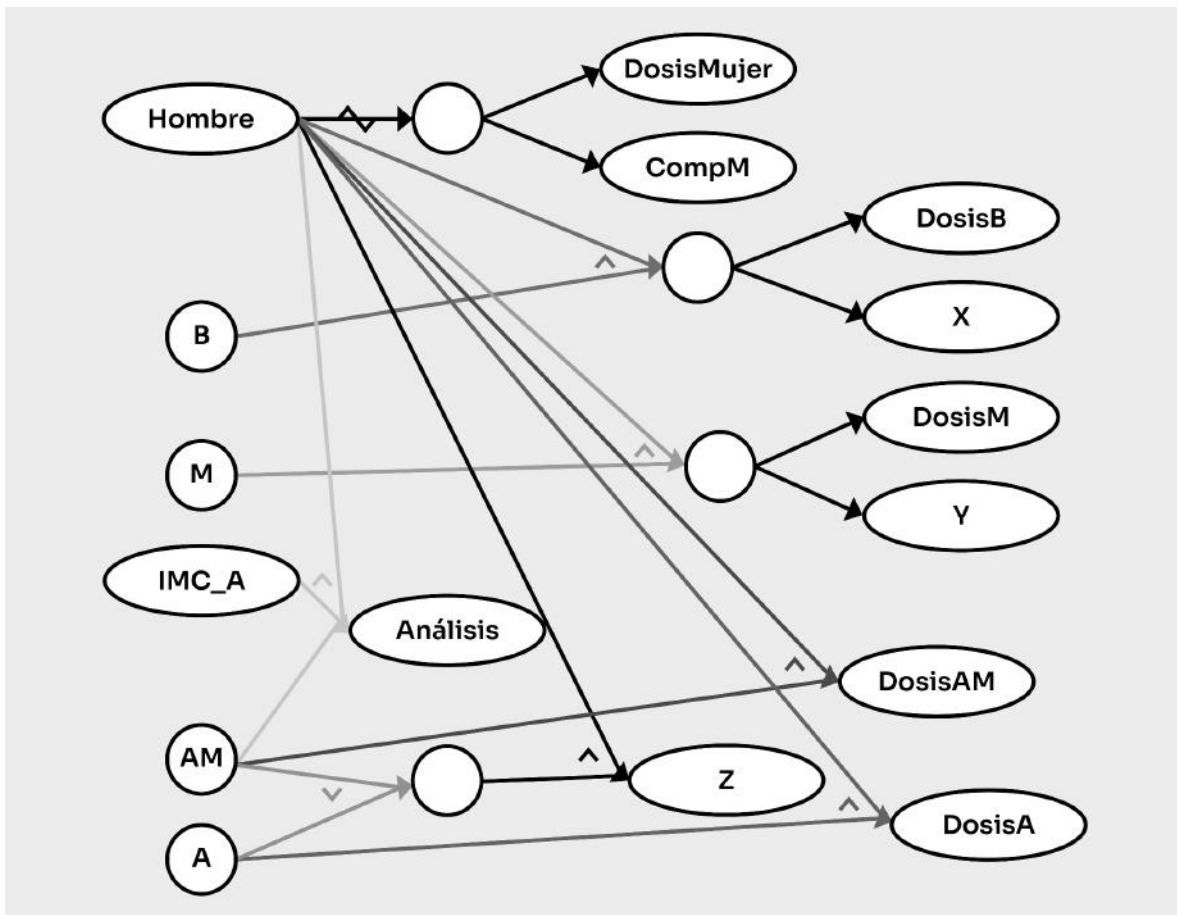
Tabla de decisión para el grafo causa-efecto

No existe una única forma de llenar las celdas de esta tabla. Lo interesante es utilizar el grafo para identificar la forma de combinar las causas para probar los distintos efectos que debe mostrar el sistema. Por ejemplo, si queremos comprobar que el sistema indica correctamente cuándo debe hacerse un análisis y cuándo no, podemos generar pruebas pensando específicamente en esa situación, y ver qué combinación de causas debe generar esa situación y cuáles no.

Para completar la tabla, se debe rellenar cada celda con 1 (indicando que se cumple la causa o efecto correspondiente), 0 (indicando que no se cumple) o con un guion “-” (indicando que no afecta o no está determinado). Conviene rellenar primero todos los 1 de acuerdo a las relaciones. Si existe una relación A->B, entonces, habrá una fila A en las causas en la que llenaremos con un 1 y habrá una fila B en los efectos que llenaremos con 1. Luego, llenaremos el resto de las celdas de esa columna con cuidado, poniendo 0 solo si: en las causas el valor no debe combinarse para que se aplique esa regla de negocio y en los efectos solo si hace falta verificar que el efecto resultante no es el de esa columna. El resto de celdas se llenan con “-”. Aquí es donde hay que tener cuidado con las relaciones entre las causas evitando inconsistencias.

Derivando los casos de prueba con Grafos Causa-Efecto

Veamos cómo usar esta técnica sobre el mismo ejemplo presentado para las tablas de decisión. El ejemplo se basa en un sistema que ayuda a determinar el tratamiento a aplicar en base a la edad, sexo y al índice de masa corporal (IMC). La siguiente figura muestra una forma de representar esta lógica con un grafo causa-efecto.



Ejemplo de grafo causa-efecto

Prestar atención a que en esta representación no se está teniendo en cuenta la relación entre las causas. Por ejemplo, si ocurre la causa B implica que no ocurre M ni A ni AM, pues son excluyentes. Para eso también está contemplada una forma de

representarlo, pero por mantenerlo simple no lo tendremos en cuenta, considerando que el *tester* prestará atención a la relación entre las causas al momento de derivar los casos de prueba.²³

Si bien la lógica del ejemplo no resulta muy compleja, ya puede verse que, ante situaciones en las que se manejan más variables y condiciones, la representación puede quedar poco práctica. En esos casos, es conveniente comenzar directamente con la representación de tablas de decisión.

Por ejemplo, en la siguiente tabla se muestra la tabla derivada a partir del ejemplo que venimos viendo. Ahí la primera columna de datos indica que si no es hombre se debe verificar que se suministre la dosis de mujer y el complemento de mujer. Luego, se indica que no se debe hacer ningún análisis, porque existe una regla que dice que el análisis se asigna solo a los hombres adultos mayores y con el IMC en el rango A. Las celdas que corresponden al IMC y a las franjas etarias se llenaron con “-”, pues no influyen en esta regla.

Causas						
Hombre	0	1	1	1	1	1
IMC_A	-	-	-	1	-	-
B	-	1	0	0	0	0
M	-	0	1	0	0	0
A	-	0	0	0	1	0
AM	-	0	0	1	0	1
Efectos						
DosisMujer	1	0	0	0	0	0
CompM	1	0	0	0	0	0
DosisB	0	1	0	0	0	0
DosisM	0	0	1	0	0	0
DosisA	0	0	0	0	1	0
DosisAM	0	0	0	-	0	1
X	0	1	0	0	0	0
Y	0	0	1	0	0	0
Z	0	0	0	-	-	1
Análisis	0	0	0	1	0	-

Tabla de decisión para el grafo causa-efecto

²³ Puede consultarse esta técnica con mayor detalle en el libro de Myers *The art of software testing*. <https://www.goodreads.com/book/show/877789.The_Art_of_Software_Testing>

Cada columna representa un caso de prueba, donde se presentan las entradas (causas) y los resultados esperados (efectos).

Algo a tener cuidado es no generar inconsistencias. En el ejemplo, no tiene sentido poner un caso de prueba en el cual se combinen las causas *Menor* y *Adulto Mayor*, ya que son excluyentes).

¿Estos casos de prueba se corresponden con los derivados con la técnica de tablas de decisión? ¿Por qué?

Técnica de Máquinas de Estado

Una máquina de estados (también conocida como *FSM*, del inglés, *finite state machine*) es un grafo dirigido cuyos nodos representan estados y cuyas aristas representan transiciones entre dichos estados. De esta forma, podemos modelar el comportamiento de entidades o los pasos de un caso de uso (como se vio en una sección anterior).

La estructura de la máquina de estados puede utilizarse para diseñar casos de prueba que la recorran, atendiendo a algún criterio de cobertura. Algunos de estos criterios de cobertura son:

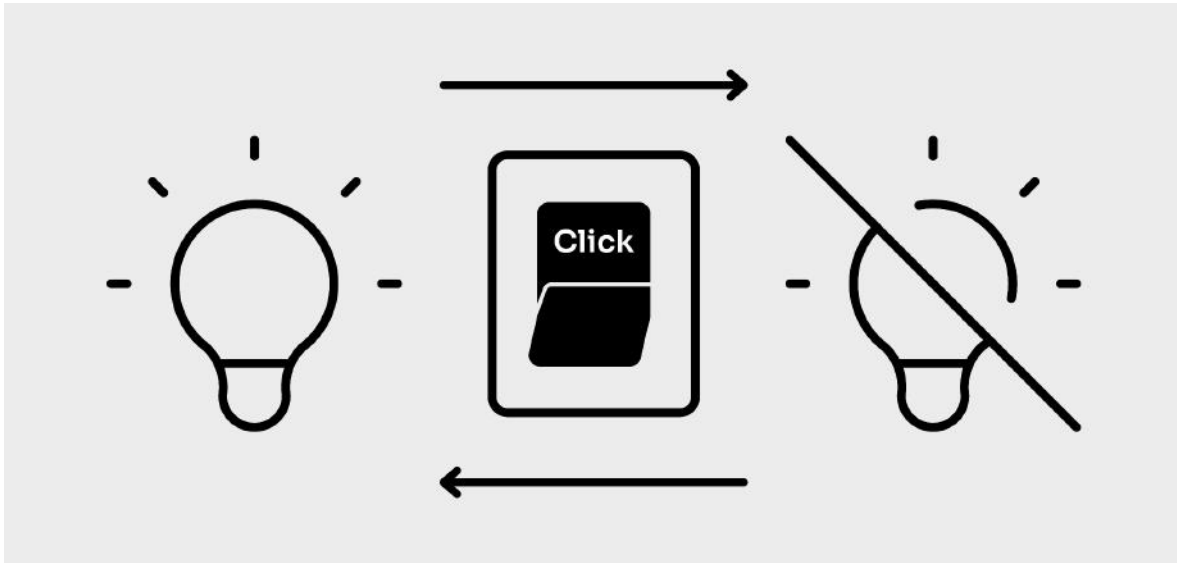
- **Cobertura de estados:** Este criterio se satisface cuando los casos de prueba recorren todos los estados.
- **Cobertura de transiciones:** En este caso cuando se recorren todas las transiciones.
- **Cobertura de pares de transiciones:** Para cada estado se cubren las combinaciones de transiciones de entrada y salida.

Este tipo de técnicas es de las más usadas en el mundo del *Model-Based Testing* (MBT), para lo que recomendamos investiguen sobre el trabajo que ha publicado Harry Robinson²⁴.

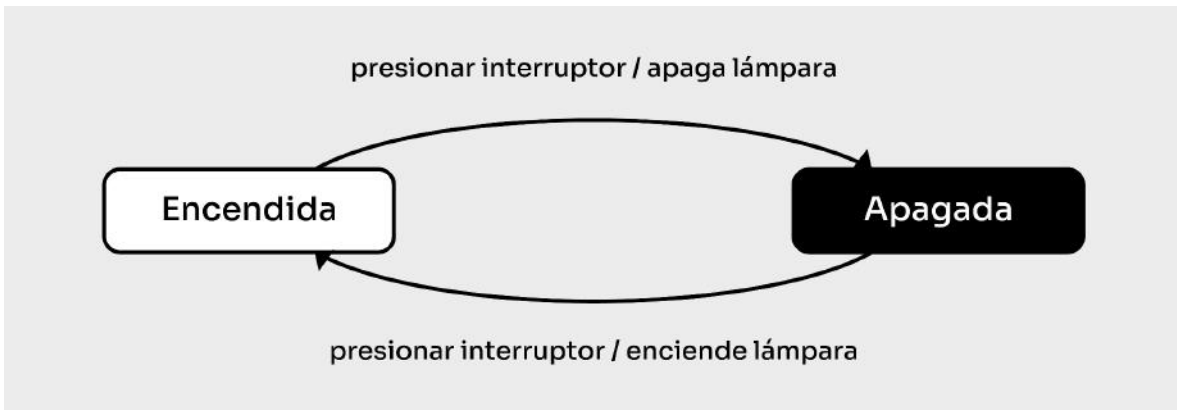
Aplicación de la técnica en un ejemplo

Veamos los componentes más en detalle y cómo se usan para representar el comportamiento de un sistema. Para ejemplificar, nos basaremos en el modelo de la siguiente figura, que representa el comportamiento de un interruptor de una lámpara. Si está apagada y se presiona, entonces, la lámpara se enciende y, si estaba encendida y se presiona, entonces, se apaga.

²⁴ Para profundizar sobre el trabajo de Harry Robinson: <<http://www.harryrobinson.net>>



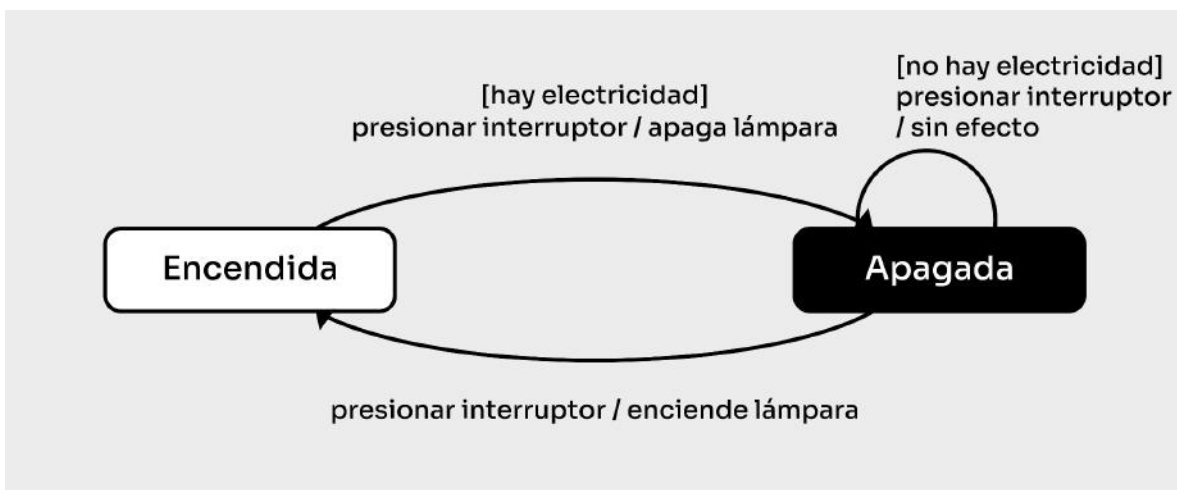
Esta situación tan cotidiana se puede representar con la siguiente máquina de estados:



- **Estado:** Los *estados* son condiciones en las que está un sistema mientras espera por uno o más eventos (encendida, apagada, etc.). El mismo evento puede ocasionar comportamientos distintos en los distintos estados (esa es la forma de distinguir los estados). Por ejemplo, el mismo evento, *presionar interruptor*, genera distintas acciones, según el estado en el que se encuentra el sistema.
- **Evento:** Es el estímulo *externo* que recibe el sistema, causa un cambio en su estado. Por ejemplo, el evento *presionar interruptor* hace que la luz cambie

de estado. Una vez que se da un evento, el sistema puede cambiar de estado, permanecer en el estado actual y/o ejecutar una acción.

- **Acción:** Una acción es una operación que se inicia a partir de un evento. Por ejemplo, luego de *presionar interruptor* se ejecuta la acción de *apagar lámpara*.
- **Transición:** Una transición está representada por una flecha y a su vez indica que se da un cambio, un pasaje de un estado a otro estado. En cada una de estas flechas podremos ver conceptos de guardas (condiciones), eventos y acciones.
- **Guarda:** Es una condición que debe darse para que el evento ocasione que se vaya por una determinada transición. En el ejemplo, podríamos agregar una transición más (otra flecha) que vaya de *apagada* a *apagada* (un lazo sobre el mismo estado), y agregar la guarda que diga: *no hay electricidad*.



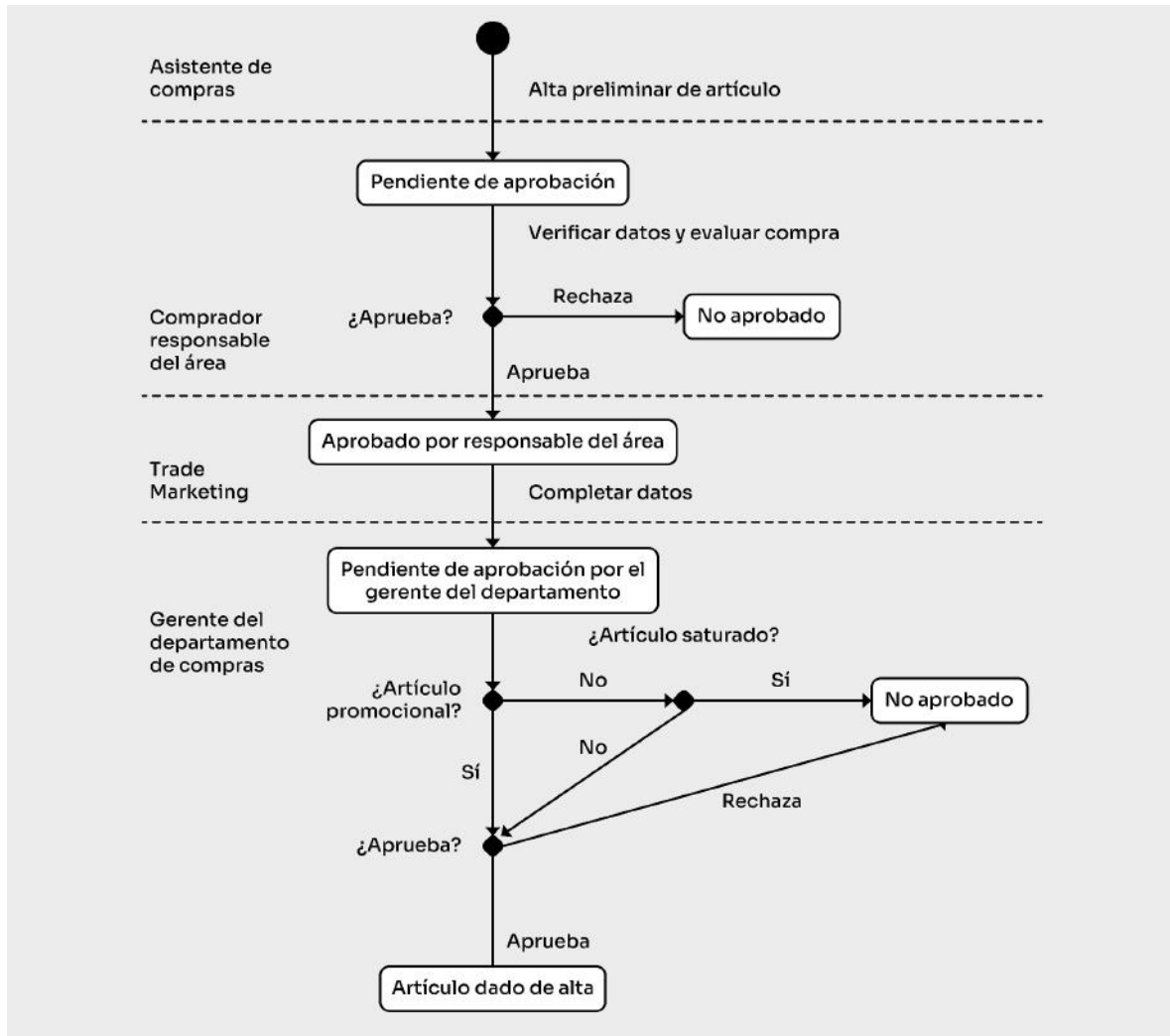
Veamos ahora cómo generar casos de prueba para esta máquina de estados según los criterios de cobertura.

- Caso de prueba 1: Lámpara apagada, sin electricidad, presiono el interruptor.
- Caso de prueba 2: Lámpara apagada, con electricidad, presiono el interruptor.
- Caso de prueba 3: Lámpara encendida, presiono el interruptor.

Observen que solo con el caso de prueba 2 ya tenemos cobertura de estados. Si quisiéramos cubrir todas las transiciones, entonces, necesitamos los casos de prueba 1, 2 y 3. Para poder cubrir los pares de transiciones, deberíamos realizar más flujos aún sobre el sistema. Esto muestra cómo un criterio subsume a otro.

Pensemos ahora en **un ejemplo un poco más complejo** o más parecido a lo que nos enfrentaremos al probar un sistema. El objetivo planteado es diseñar los casos de prueba para la creación de artículos en un sistema, teniendo en cuenta todo el ciclo de vida y las distintas personas que tienen que realizar diferentes validaciones y acciones sobre la información cargada al sistema. Para eso, decidimos modelar el comportamiento de los artículos con una máquina de estados y, a partir de ella, derivar los casos de prueba. Los artículos pueden tener distintos estados y el comportamiento esperado para cada acción o funcionalidad del sistema dependerá del estado asociado al artículo.

La siguiente figura presenta una máquina de estados parcial para el concepto *artículo*. A la izquierda de la figura se puede ver el actor involucrado en la tarea y, en este caso puntual, las guardas están representadas como puntos de decisión en el diagrama.



Máquina de estados para sistema de gestión de artículos

Los estados determinan distintos comportamientos de las instancias de artículos. Las transiciones se corresponden con funcionalidades del sistema que hacen que el artículo cambie su comportamiento, su estado.

La máquina de estado muestra el comportamiento esperado de la entidad *artículo* en su ciclo de vida. Lo que se intentará es cubrir todas las transiciones y nodos de este diagrama. El resultado es un conjunto de secuencias a seguir sobre la máquina de estados, que definen los casos de prueba interesantes, que para esta máquina de estados son las siguientes:

- Caso 1: Alta preliminar de artículo, verificar datos y rechazar compra.
- Caso 2: Alta preliminar de artículo, verificar datos y aprobar por el comprador responsable de área.
- Caso 3: Alta preliminar de artículo, verificar datos y aprobar por el comprador responsable de área, completar los datos por *trade marketing*, el gerente de departamento de compras no aprueba la compra por ser un artículo no promocional y saturado.
- Caso 4: Alta preliminar de artículo, verificar datos y aprobar por el comprador responsable de área, completar los datos por *trade marketing*, el gerente de departamento de compras no aprueba la compra para un artículo promocional y saturado.
- Caso 5: Alta preliminar de artículo, verificar datos y aprobar por el comprador responsable de área, completar los datos por *trade marketing*, el gerente de departamento de compras aprueba la compra para un artículo no promocional y no saturado.
- Caso 6: Alta preliminar de artículo, verificar datos y aprobar por el comprador responsable de área, completar los datos por *trade marketing*, el gerente de departamento de compras aprueba la compra para un artículo promocional y saturado.

Hasta aquí, se cuenta con los casos de prueba en alto nivel (casos de prueba abstractos). O sea, solo se cuenta con las secuencias de funcionalidades que se deben invocar y las condiciones sobre los datos. Luego, se deben definir los datos concretos que permitan ejecutar sobre el sistema. Al ejecutar, se utiliza la máquina de estados como oráculo (para determinar si el resultado es válido o inválido), pues se debe verificar en cada paso si se llegó al estado esperado o no. Cada caso de prueba abstracto podrá corresponderse con uno o más casos de prueba concretos en base a los distintos datos que se seleccionen.

Herramientas para derivar pruebas con Máquinas de Estado

Si bien esta técnica puede usarse en forma manual, como se acaba de mostrar, existen herramientas que facilitan la aplicación de la técnica. A continuación, mostraremos cómo uno podría utilizar GraphWalker²⁵ para esto con el fin de entender la ventaja que nos brindan las herramientas.

GraphWalker nos permite diseñar la máquina de estados en formato gráfico y generar automáticamente los casos de prueba para cubrir el modelo en base a diferentes criterios. Para utilizarlo, recomendamos revisar el manual de uso, porque al momento en que lo usamos no nos resultó tan intuitivo.

Una vez generado el modelo, es posible descargarlo en formato JSON²⁶ y con esto, la aplicación de GraphWalker nos genera una serie de caminos según el criterio de cobertura elegido. Para todas las opciones y detalles, recomendamos revisar la documentación del producto²⁷. Además de esto que comentamos acá, la herramienta brinda muchísimas posibilidades más, también útiles para la ejecución de pruebas de forma automatizada.

A modo de ejemplo, la siguiente es una salida del programa aplicada sobre la máquina de estados de la lámpara, donde se puede ver la secuencia de pasos que sugiere seguir para poder cubrir todas las aristas.

²⁵ GraphWalker: <<https://github.com/GraphWalker>>

²⁶ JSON: <<https://www.json.org>>

²⁷ Documentación GraphWalker:
<<https://github.com/GraphWalker/graphwalker-project/wiki/Offline>>

```
{"currentElementName":"Apagada"}  
{"currentElementName":"presionar interruptor / enciende lámpara"}  
{"currentElementName":"Encendida"}  
{"currentElementName":"[hay electricidad] presionar interruptor / apaga lámpara"}  
{"currentElementName":"Apagada"}  
{"currentElementName":"[no hay electricidad] presionar interruptor / sin efecto"}  
{"currentElementName":"Apagada"}
```

Los distintos niveles de cobertura nos pueden garantizar mayor o menor grado de pruebas. Entonces, para decidir qué cobertura utilizar en cada caso hay que “poner en la balanza” la criticidad de la funcionalidad probada y el tiempo disponible.

Técnica de Matriz CRUD

Cada entidad de un sistema de información tiene un ciclo de vida: todo nace, crece (se actualiza), muere (se elimina) y en el medio es consultado. A esto se suele referenciar como *el patrón CRUD* (crear, leer, actualizar, eliminar, del inglés: *create, read, update, delete*).

Para tener una vista general sobre el ciclo de vida de las entidades y cómo es afectado en las distintas funcionalidades que estamos probando, podemos armar una *CRUD Matrix*. Esta tiene como columnas las distintas entidades que interesa analizar y, como filas, las funcionalidades del sistema. En cada celda luego se pone una C, R, U y/o D según la operación que se realice sobre la entidad en la funcionalidad correspondientes a su fila y columna.

Imaginemos que tenemos un sistema que maneja clientes, facturas y productos. Para ejemplificar el análisis CRUD con este sistema veamos la siguiente tabla. Por ejemplo, en la funcionalidad *New Invoice* (para la creación de una nueva factura), se puede ver que hay una lectura y actualización en la entidad *Client*, pues se leen los datos del cliente para poder seleccionar a quién corresponde la factura y se actualiza el balance. Lo mismo con *Product*, pues se muestran los datos de los productos y se actualiza su *stock*.

Ya con esto se puede hacer una verificación estática muy interesante, que es verificar la completitud: ver que las cuatro letras aparezcan en cada columna. Si falta una operación en alguna entidad, no indica necesariamente que sea un error, pero al menos llama la atención como para que se verifique por qué no está disponible esa operación en ninguna funcionalidad. La aclaración es porque pueden existir datos maestros que no se puedan crear, sino que se dan de alta por un sistema externo, directamente por un administrador de la base de datos o hay datos que no se deben eliminar nunca, etc..

	Client	Cities	Countries	Invoice	Products	Prices
<i>New Client</i>	C	R	R			
<i>New City</i>		C	R			
<i>New Country</i>			C			
<i>New Invoice</i>	R, U			C	R, U	R
<i>New Product</i>					C	C, R, U
<i>List Clients</i>	R	R	R			
<i>List Cities</i>		R	R			
<i>List Countries</i>			R			
<i>List Invoices</i>	R			R		
<i>List Products</i>					R	
<i>Update Clients</i>	U	R	R			
<i>Update Cities</i>		U	R			
<i>Update Country</i>			U			
<i>Update Invoice</i>	R, U			U	R, U	
<i>Update Product</i>					U	C, R, U, D
<i>Delete Clients</i>	D	R	R			
<i>Delete Cities</i>		D	R			
<i>Delete Countries</i>		R	D			
<i>Delete Invoices</i>	R			D	R	R
<i>Delete Products</i>					D	D, R

Matriz CRUD para sistema de facturación de ejemplo

Luego, se pasa a realizar una verificación de consistencia: probar distintas funcionalidades de forma tal que se haga pasar por todo el ciclo de vida a cada entidad. Esto es, armar casos de prueba para cada entidad de forma tal que comiencen con una C, sigan con cada posible U y terminen con una D. Luego de cada una de estas acciones, se debe agregar al menos una acción de R (lectura). Esto es para verificar que el procesamiento fue realizado correctamente y no hay algo inconsistente o datos corruptos.

Para cada entidad relevante deberían cubrirse todas las C, R, U y D de cada función, de forma de considerar cubierto el criterio. Por ejemplo, podríamos considerar probar el siguiente ciclo funcional:

1. *New Client.*
2. *New Invoice.*
3. *List Clients.*
4. *Delete Client.*

Con este caso de prueba, estamos cubriendo CRUD para la entidad *Client*. Se deberían probar todas las entidades con estas consideraciones.

Para ir más lejos (logrando una mayor cobertura), se podría ejecutar toda funcionalidad que contenga una R, luego de cada función que tenga una U para esa entidad, a modo de verificar que cada vez que un dato se actualice en algún lugar se refleje en cada pantalla donde sea consultado.

Comentarios finales del capítulo

¿Con esto ya estamos listos para probar lo que sea? Bueno, tenemos una base, pero siempre es importante investigar más técnicas y tenerlas presentes como para que en cada nueva situación tengamos la capacidad de seleccionar la más adecuada para el contexto que nos toca probar (a modo de repaso y muy en alto nivel: las tablas de decisión son útiles cuando hay reglas de negocio definiendo cierta lógica, las máquinas de estado para cuando tenemos entidades que cambian el comportamiento según su estado, la matriz CRUD para ciclo de vida de entidades, la de grafos causa-efecto para cuando hay reglas del tipo *if-then-else*).

¿Cómo seguir?

- Practicar estas técnicas con casos reales, con aplicaciones que usen a diario, que prueben a diario.
- Investigar otras técnicas. Existen cientos de fuentes de donde obtener casos con ejemplos, experiencias prácticas.
- *Pair testing*: trabajar en conjunto con otro *tester* e intercambiar técnicas, mostrar cómo cada uno aplica cada técnica y discutir sobre cuál es la mejor forma para lograr los mejores resultados.

Estaremos encantados también de recibir sus inquietudes o comentarios sobre estas técnicas u otras que sean de su agrado.

No he contado la mitad de lo que vi.

Marco Polo

INTRODUCCIÓN AL *TESTING* EXPLORATORIO

Primero que nada, debemos aclarar que el objetivo de este capítulo es presentar la estrategia *Testing* Exploratorio para tener una perspectiva más amplia sobre el *Testing* Funcional. En segundo lugar, tal como lo mencionamos, el *Testing* Exploratorio es una estrategia, un acercamiento a nuestro trabajo como *testers* en el que podemos conjugar todas las técnicas de *Testing* Funcional que conozcamos y que apliquen dentro del contexto de nuestro proyecto. Veremos entonces una introducción a esta temática para que quien lea, luego, pueda profundizar con un panorama de lo que se incluye en las siguientes páginas.

¿Qué es el *Testing* Exploratorio?

Veamos algunas definiciones provistas por algunas de las personas consideradas gurúes en el tema. En el año 1983, Cem Kaner le ponía un nombre a esta metodología de trabajo y su definición es la siguiente:

Es un estilo de *testing* de *software* que enfatiza la libertad personal y la responsabilidad del *tester* como individuo, para que de forma continua se optimice el valor de su trabajo, al tratar tanto tareas como el aprendizaje, diseño de pruebas y su ejecución, como actividades que se apoyan mutuamente y que se ejecutan en paralelo a lo largo de un proyecto.

Por otro lado, tenemos otras definiciones provistas por James Bach (2003) más populares, como las siguientes: “El *Testing* Exploratorio consiste en el aprendizaje, diseño y ejecución de pruebas de forma simultánea.”

Las visiones más actualizadas y recomendadas para profundizar en esta temática vienen de la mano de Elizabeth Hendrikson y su libro *Explore it!* (2012) o de todo el trabajo que siempre publica Maaret Pyhäjärvi, por ejemplo su libro *Exploratory testing* (2017).

A partir de las definiciones, y tal como lo veremos a lo largo de este capítulo, el *Testing* Exploratorio presenta una estructura que es sencilla de describir. Durante un lapso el equipo de *testing* interactúa con un producto para cumplir con el objetivo de una misión y posteriormente presentar y reportar los resultados que el resto de los actores del proyecto utilizarán para tomar decisiones. Por lo tanto, con **una misión** se describe qué es lo que pretende probar del producto o sistema bajo pruebas, cuáles serán los tipos de incidentes que se buscarán, lo cual ayudará a dar forma a **una estrategia**, y por último, aunque no por ello menos importante, los riesgos involucrados. La misión, la puede diseñar el *tester*, el equipo de *testing* o también ser asignada por el líder de *testing*.

Los elementos básicos que componen una misión son: tiempo, objetivos y reportes. Tal como lo mencionamos, dicha aparente simplicidad permite desplegar una amplia gama de posibilidades para la aplicación del *Testing Exploratorio*.

Es recomendable, al trabajar con este tipo de estrategia, tomar notas sobre todo lo que se hizo y lo que se observa que sucede durante el transcurso de nuestras pruebas. También cabe destacar que este tipo de enfoque puede aplicarse en cualquier situación donde no sea evidente cuál es la próxima prueba que se debe ejecutar en el sistema, cuando se requiere obtener una rápida retroalimentación, ya sea de un producto o funcionalidad, o si queremos aprender y aislar un defecto en particular, entre otros posibles escenarios.

Construyamos nuestro mapa

Compartimos con ustedes una definición del concepto de *exploración*, (Keay, 1994) que expresa lo siguiente:

Por lo tanto, para calificar como exploración un viaje tenía que ser creíble, tenía que involucrar privaciones y riesgos, y tenía que incluir la novedad del descubrimiento. Después de eso, al igual que el cricket, era algo un tanto complejo de explicar para el no iniciado. Pero un elemento era absolutamente vital; verdaderamente era lo que exactamente distinguía la era de la exploración de eras previas de descubrimiento, las cuales necesitaban de la adopción de la palabra ‘exploración’. Era, sencillamente, una reverencia por la ciencia.

Inicialmente un explorador que es enviado a construir un mapa, aunque tenga una idea “macro”, desconoce el territorio. Entonces, comienza a explorar, a tomar apuntes de sus hallazgos y conclusiones, y así va mejorando el aspecto del mapa. Un explorador debe avanzar estando alerta a lo que pueda surgir, debe caminar teniendo su mente abierta y receptiva.

Para elaborar mapas más precisos, es importante tomar notas de lo que se va descubriendo, ya sea en forma esquemática o en forma tabulada. Los exploradores tenían a su disposición una variedad de herramientas que los ayudaban en su actividad, por ejemplo:

- Brújula, para indicar en qué dirección iban.
- Catalejo, para enfocar a lo lejos y “acercar” los lugares.
- Compás, que permitía medir y ubicar con más precisión los sitios explorados.

Todo explorador necesita una misión, conocimientos, herramientas y experiencia para que pueda hacer inferencias y conjeturas a partir de los resultados obtenidos con los recorridos que haya hecho anteriormente.

Tal como alguien que explora, alguien que hace *testing* construye un modelo mental que está relacionado con el comportamiento del producto. Es decir, a medida que aprende con los sucesivos recorridos, también irá diseñando nuevas pruebas, nuevos caminos alternativos, utilizando las herramientas que mejor se ajusten cada vez, como por ejemplo, una técnica de *Testing* Funcional que no haya utilizado hasta el momento. A lo largo del proceso de descubrimiento, aprende del sistema, de su comportamiento y también de las pruebas realizadas, donde a su vez, las pruebas sucesivas se basan en las efectuadas anteriormente.

A semejanza del explorador, un buen *tester* exploratorio mantiene una sana dosis de escepticismo, que lo motiva y le recuerda que debe registrar sus hallazgos, así como ideas de *testing* para que él y sus colegas puedan utilizar en otra ocasión que lo requiera. Despliega todas sus habilidades, se interroga sobre la credibilidad de los resultados, utiliza las herramientas disponibles e investiga nuevas.

Características del *Testing* Exploratorio

Ahora que hemos visto algunas definiciones, veamos las principales características del *Testing* Exploratorio. Con ellas, entenderemos mejor dónde nos puede aportar mayor valor:

1. Las pruebas no son definidas con anticipación, en este contexto se espera que el *tester* aprenda con rapidez y velocidad acerca de un producto o nueva funcionalidad, mientras que se provee retroalimentación al resto del equipo.

2. Los resultados obtenidos durante pruebas anteriores guiarán las acciones, los pasos y los siguientes escenarios de prueba a ejecutar, construyendo así un conjunto de pruebas más eficaz.
3. Su foco está en encontrar problemas y defectos por medio de la exploración del sistema bajo pruebas.
4. Es un enfoque de *testing* que consiste en una serie de actividades que se realizan en simultáneo, tales como el aprendizaje del sistema, diseño y ejecución de las pruebas.
5. La efectividad del *testing* se apoya en el conocimiento, habilidades y experiencia del *tester*.

Estilos de *Testing* Exploratorio

Veremos ahora algunos de los distintos “sabores”, por así llamarlos, que puede tener el *Testing* Exploratorio. Fueron extraídos del libro *Exploratory software testing*, de James Whittaker (2011). Luego, vamos a profundizar en una sección aparte en uno de esos sabores, en el conocido como *Testing Exploratorio Basado en Sesiones*, ya que es el que le brinda una mejor estructura, facilidad de implementación, de aprendizaje y sentido de responsabilidad. Además, es más simple presentar resultados a los actores involucrados.

Ad-hoc testing

De acuerdo al diccionario de la Real Academia Española, se define *ad-hoc*:

1. Para referirse a lo que se dice o hace solo con un fin determinado.
2. Adecuado, apropiado, dispuesto especialmente para un fin.

En nuestra línea de trabajo, el término se emplea en el sentido de buscar una solución particular para un problema, sin hacer un esfuerzo de abstracción y sistematización de la respuesta. James Whittaker presenta el estilo *Ad-hoc* o *libre* como una exploración *ad-hoc* de las funcionalidades de una aplicación, sin

seguir un orden determinado ni preocuparse por cuáles fueron o no cubiertas. No se aplican reglas ni patrones para hacer las pruebas, simplemente se prueba.

Este estilo podría aplicarse ante la necesidad de tener que realizar una prueba de humo rápida, con la que podamos ver si hay o no caídas mayores. También podrá aplicarse para tener un primer contacto con la aplicación antes de pasar a una exploración con técnicas más sofisticadas. Este es un ejemplo muy clásico, revisar una nueva aplicación o una nueva versión o una nueva funcionalidad. En base a esta exploración *ad-hoc* la persona que está explorando y probando irá aprendiendo, generando sus primeras preguntas, su mapa mental inicial, y de ahí podría pasar a otro tipo de exploración más refinada y documentada para cada una de las secciones del producto que identificó por ejemplo.

Por lo tanto, como estilo de exploración, no requiere una gran preparación, por lo cual no debería despertar muchas expectativas. El resultado de aplicar el estilo libre consiste solamente del reporte de los incidentes detectados y se debe tener en cuenta que los caminos transitados y los incidentes no evidenciados se pierden.

Testing Exploratorio Basado en Estrategias

Si combinamos la experiencia con la percepción de un buen *tester* exploratorio y con las técnicas conocidas para detectar incidentes (como por ejemplo valores límite y particiones de equivalencia), entonces, las pruebas que se ejecuten sobre un sistema se ven potenciadas. Para poder hacer un uso efectivo de este modelo, es importante contar con un repertorio de técnicas que sea lo más amplio posible, sumado a ideas, creatividad y experiencias de prueba anteriores. Por esto, se dice que esta técnica de *Testing* Exploratorio está basada en estrategias, ya que su éxito dependerá de la aplicación de técnicas existentes de derivación de casos de prueba, pero al momento de estar ejecutándolas.

Creemos que en el proceso natural de formación en *testing*, en determinado momento, comenzamos a diseñar pruebas basándonos en estrategias existentes, pero lo hacemos quizá en forma inconsciente. Tal vez pensemos en los valores límites o en partición de equivalencias, pero sin formalizarlo, sino que se piensan las fronteras que hay entre las clases de equivalencia y se seleccionan valores de esas fronteras en forma natural. O si detectamos que hay distintos comportamientos en base al estado de la aplicación, quizá queramos ejecutar cada

uno de los estados, aplicando así inconscientemente una técnica basada en máquinas de estado.

Es necesaria mucha experiencia en técnicas de diseño como para poder aplicar esta estrategia. Seguramente, al llegar a ese nivel de manejo de las técnicas, esta técnica de *Testing* Exploratorio Basado en Estrategias sea lo que hacemos generalmente al interactuar con cualquier nueva aplicación y, con seguridad, la utilizamos mezclada con cualquiera del resto de las técnicas exploratorias.

Testing Exploratorio Basado en Sesiones

Las sesiones de *Testing* Exploratorio surgieron como parte de un trabajo conjunto entre James y Jonathan Bach, para reinventar la gestión de ese tipo de *testing*. A partir de sus observaciones, pudieron notar que los *testers* hacían varias cosas que no estaban relacionadas al *testing*. Por lo tanto, con el propósito de registrar esas actividades, era necesario contar con una manera de distinguirlas del resto, y así, surgieron las sesiones.

En la práctica de *Testing* Exploratorio, una sesión es la unidad básica de trabajo de *testing*. Lo que se conoce como una sesión es un esfuerzo de *testing* dentro de un bloque de tiempo **ininterrumpido**, **revisable** y con una **misión**. Decir que tenga una **misión** quiere decir que cada sesión tendrá un objetivo, o sea, lo que estemos probando o los problemas que estemos buscando. Por **ininterrumpida**, queremos decir que no tendrá interrupciones significativas, sin correos electrónicos, reuniones, llamadas telefónicas, etc. Por último, que sea **revisable** quiere decir que contará con un reporte, algo que llamaremos la hoja de la sesión, que proveerá información sobre lo ocurrido durante la sesión.

De esta forma, las sesiones son debidamente documentadas y facilitan la gestión de las pruebas y la medición de la cobertura.

Cada misión se organiza en sesiones relativamente cortas, con duraciones que varían entre los 45, 90 y 120 minutos. Esto es principalmente con el propósito de fijar la atención en la prueba en forma sostenida y que ningún aspecto nos pase desapercibido. Evitamos así que se nos pasen cosas, ya sea por falta de atención o

porque nuestra atención queda presa de lo que ya conocemos y perdemos capacidad de asombro.

Cada sesión tiene una única misión u objetivo definido.

Estas sesiones son establecidas por el *tester* o le son asignadas por el líder de *testing*. Periódicamente, se revisan las anotaciones de las sesiones, ya que puede suceder que surjan nuevas misiones y que alguna haya quedado obsoleta. También se puede identificar riesgos que no fueron considerados originalmente y desestimar otros en el proceso.

El cubrimiento se mide por la cantidad de sesiones por misión, las áreas exploradas por las sesiones, su duración y los incidentes detectados. Como se puede apreciar, el *Testing* Exploratorio cuenta con una estructura, exige una preparación importante y requiere de una planificación.

La estructura con la que cuenta el *Testing* Exploratorio, y particularmente trabajando con sesiones, no es procedural, sino sistemática, puesto que se vale de varias fuentes tanto para su planificación, como para la actualización de su planificación. También recordemos que, como lo definió Cem Kaner, este acercamiento al *testing* enfatiza la responsabilidad y la libertad de *testers*, por lo tanto, el *tester* exploratorio debe poder construir una historia que brinde esencia a su trabajo.

Trabajando con *Testing* Exploratorio Basado en Sesiones

Trabajar con una estrategia exploratoria puede significar un esfuerzo amplio y abierto, para ello, es necesario contar con un mecanismo como las sesiones, que brinde estructura y organización. De lo contrario, podríamos invertir horas o incluso días sin obtener información que pueda ser de utilidad para tomar decisiones sobre el sistema que se haya estado probando.

Hemos hablado de las sesiones, de algunas de sus características, por ejemplo, que deben tener una misión, pero ¿cómo podríamos crear una sesión si debiéramos hacerlo? ¿Qué deberíamos tener en cuenta?

Algunos puntos a considerar al armar una sesión serían los siguientes: **dónde** debemos enfocar nuestros esfuerzos durante nuestra exploración; **qué** recursos tenemos a nuestra disposición; **cuál** es la información que descubrimos durante nuestro trabajo.

Dónde: En este punto, necesitamos saber y también comprender cuál es el objetivo de nuestro trabajo de exploración. Podría ser posible que estemos trabajando probando un requerimiento, un módulo, una nueva funcionalidad, etc.

Qué: ¿Vamos a contar con herramientas para asistir a mi trabajo? ¿Qué tipo de herramientas? Algo a tener en cuenta, es que las herramientas pueden estar en cualquier forma, desde una nueva técnica, la ayuda de un compañero u otro actor que esté involucrado en el proyecto, como por ejemplo, un analista de negocio o quizás la configuración de un sistema, entre otros.

Cuál: Una vez que contamos con los puntos anteriores, debemos enfocarnos en encontrar información relevante, información que nos ayude a proporcionar valor

al resto del equipo. ¿Sobre qué aspecto queremos encontrar y proveer valor encontrando información? ¿Sobre seguridad, *performance*, usabilidad?

Veamos un ejemplo de misión que usaríamos en una sesión. Está pensada con una orientación a encontrar problemas relacionados con la seguridad de una aplicación:

“Realizar ataques por inyección SQL²⁸ para descubrir potenciales vulnerabilidades de seguridad.”

Como podemos ver, la misión es muy específica, ya que trata sobre realizar ataques que hagan foco en inyecciones SQL. A continuación, veremos algunas recomendaciones para escribir buenas misiones.

Escribiendo una buena misión

En primer lugar, una buena misión debería ser capaz de proveer de dirección a quien la lea sin restringir las acciones o tareas de *testing* a realizar. Es decir, el *tester*, grupo de *testers* que trabajarán en conjunto a lo largo de la sesión, o incluso la gerencia, deberían ser capaces de comprender cuál fue el objetivo que se tenía en mente cuando la revisen.

Ejemplo:

“Explorar la edición de los apellidos que concuerden con Pérez y editarlos para comprobar que soportan tildes.”

Esta misión es demasiado específica, por lo que es altamente probable que el *tester* invierta una gran cantidad de tiempo documentando pruebas sin mucho beneficio.

Por otro lado, las misiones que son demasiado amplias corren el riesgo de no proporcionar foco, lo que hará que el *tester* tenga mayores inconvenientes a la hora de saber cuándo terminar de explorar su objetivo.

²⁸ Inyección SQL: Técnica de ataque cibernético donde se manipulan consultas de bases de datos a través de entradas de usuario para acceder o modificar información no autorizada.

“Explorar la performance de la aplicación con todas las herramientas que se puedan encontrar.”

Esta última misión es demasiado vaga y amplia en su objetivo. Si analizamos lo que pide, requiere llevar adelante la exploración de un sistema entero, sin considerar la cobertura ni tampoco los recursos, puesto que pide hacerlo con todas las herramientas que se puedan encontrar. Con una misión de ese estilo, podríamos pasar largas jornadas frente a una computadora investigando, y aún así, no lograr seguridad de que lo que hayamos encontrado sea de utilidad para revelar riesgos, vulnerabilidades, oportunidades de mejora o defectos.

Entonces, ¿cuál debería ser la estrategia a usar para armar una buena misión que contemple el **dónde**, **qué** y **cuál**?

Podríamos comenzar aplicando un proverbio popular que dice: “Divide y vencerás”. Aquí nos resultará de utilidad la aplicación de una técnica conocida como *División Estructural de Trabajo*, o **WBS**, por sus siglas en inglés para *Work Breakdown Structure*. Esta técnica consiste en tomar una tarea grande y dividirla en unidades que sean más pequeñas, haciendo que las mismas sean manejables y fáciles de concretar en una sesión o en breves sesiones. De este modo, podemos confeccionar una misión y trabajarla en una sesión, o en un conjunto de sesiones, donde cada una haga foco en un área o funcionalidad de un sistema a probar.

Una buena misión no solo nos sirve de inspiración al momento de llevar adelante nuestras actividades de *testing*. También contienen información que provee valor al resto de los actores del proyecto.

Anatomía de una sesión

Una sesión debe contar con tres grandes características: ser **revisable**, **ininterrumpida**, y contar con un **objetivo**. Lo que veremos a continuación, se corresponde a la anatomía de una sesión de *Testing* Exploratorio para apuntar a cumplir con esas características (ver la siguiente figura).



Anatomía de una sesión de *Testing Exploratorio*

Debemos aclarar que cada una de las partes que componen la anatomía de una sesión se corresponde entre sí por orden de procedencia, tal como lo veremos a continuación.

1. **Diseño y ejecución de pruebas:** Las áreas que conforman esta parte de la anatomía de una sesión representan la exploración del producto, de una funcionalidad o nueva área dentro de la aplicación bajo prueba.
2. **Investigación y reportes de defectos:** Aunque el nombre es bastante intuitivo, de todos modos vale aclarar que corresponde a toda actividad por medio de la cual la persona, probando, encuentra un comportamiento que podría significar un problema.
3. **Armado de la sesión:** Son todas aquellas tareas de las que participa la persona que prueba, que hacen posibles los dos puntos anteriores. Por ejemplo, bajo esta categoría podría incluirse la configuración de una conexión a una base de datos; encontrar materiales, documentos y/o especificaciones que puedan servir como fuente de información; escribir la sesión de *Testing Exploratorio*, etc.

Componentes de una sesión

Los puntos que veremos a continuación forman parte de la documentación de una sesión. Primero se presenta cada una de ellas y luego se presenta un ejemplo de cómo documentar una sesión incluyendo cada una de estas partes.

1. Objetivos

Sugieren qué se debería probar y qué problemas buscar, ya sea de un producto o funcionalidad, durante un tiempo finito definido para la sesión.

Los **objetivos** forman parte tanto de **sesiones generales** como también de **sesiones específicas**, las cuales describiremos brevemente a continuación.

- **Misiones generales:** Este tipo de misión puede utilizarse, ya sea al inicio de las actividades de *testing* para aprender del sistema o aplicación bajo prueba, como también para probar y aprender sobre el funcionamiento e interacción de nuevas funcionalidades que hayan sido liberadas en una nueva versión de la aplicación.
- **Misiones específicas:** A diferencia de la anterior, este tipo de misión procura brindar un mayor foco al momento de realizar nuestras pruebas. Sin embargo, requieren de un poco más de tiempo para su escritura y es importante que ese tiempo se tenga en cuenta en el **armado de la sesión**.

2. Áreas

Esta subsección aplica tanto a misiones generales como para las específicas y su propósito es tanto contener como proveer información sobre el cubrimiento de las áreas funcionales, las plataformas, datos, operaciones, y también, técnicas de *testing* a usar para el modelado de la aplicación o sistema bajo pruebas.

3. Inicio

En este punto se procede a detallar tanto la fecha como la hora en la que se dio inicio con las actividades de *testing*.

4. *Tester(s)*

En este apartado se incluye el nombre del *tester* o *testers* que participarán y estarán a cargo de la sesión. Dos pares de ojos no solo encontrarán más *bugs*, sino también más información interesante para brindar al resto de los miembros del equipo.

5. Estructura de división de tareas

La idea de esta sección es indicar la **duración** y cómo se distribuye ese tiempo. Como se mencionó anteriormente, todas las sesiones son limitadas en tiempo, por lo que en este punto se indica el tiempo a invertir. No tiene que ser una medida exacta, ya que se procura invertir una mayor proporción del tiempo en probar el sistema bajo pruebas que en su administración. Por esto se expresa en franjas y se sugiere considerarlas de la siguiente manera:

- **Corta:** > 30 y <= 45 minutos.
- **Media:** > 45 y <= 90 minutos.
- **Larga:** > 90 y <= 120 minutos.

Luego, para indicar cómo se distribuye aproximadamente ese tiempo en las diversas tareas, se indican con porcentajes con respecto al tiempo indicado en la duración. Las tareas para las que se indicarán estos porcentajes son:

- **Diseño y ejecución de pruebas.**
- **Investigación y reporte de defectos.**
- **Armado de la sesión.**

Esto nos proporciona información que nos será de gran utilidad, por ejemplo, para ver cualitativamente cuánto tiempo se invirtió en *testing*. En ocasiones, algunas actividades que están por fuera del objetivo de la misión nos hacen comprender que necesitamos cambiarlo para sacar el mayor provecho durante nuestra sesión de *Testing Exploratorio*.

También tendremos un apartado indicando **objetivo vs oportunidad**. Ya hemos definido cuál es el **objetivo** y cómo se usa en una sesión, pero, ¿qué es la **oportunidad**? En este concepto están comprendidas todas aquellas tareas de

testing, pruebas, ejecuciones que se hayan realizado dentro de la sesión que están por fuera del **objetivo** de la misión. Habitualmente, la **oportunidad** nos servirá para pensar y definir nuevas pruebas, e incluso, para revisar el **objetivo** de la sesión sobre la que estemos trabajando. Pensemos, por ejemplo, que de una sesión cuya duración es 120 minutos pasamos la mitad de ese tiempo en la **oportunidad** porque encontramos más defectos, o problemas con el *build* de la aplicación o sistema bajo pruebas, si ese es el caso, entonces definitivamente nos va a convenir revisar el objetivo de nuestra sesión.

6. Archivos de datos

Esta sección se utiliza para detallar todas las fuentes de información que hayamos usado durante nuestras pruebas, entiéndase, documentación disponible sobre el sistema bajo pruebas, archivos de *mockups*, especificaciones, consultas SQL, capturas de pantalla, entre otros. Algo a tener en cuenta es que los archivos que aquí se detallan deberían quedar almacenados en una ubicación predefinida y conocida.

7. Notas de pruebas

Como lo indica el título de esta sección, es aquí donde el *tester* registrará las pruebas y actividades de *testing* ejecutadas a lo largo de una sesión. Las notas tienden a ser más valiosas cuando incluyen la motivación para una prueba dada. También es importante que la actividad se registre, puesto que las **notas** son el núcleo de la **sesión**. Al término de una sesión, las notas serán utilizadas para la construcción de nuevas pruebas. Además, se usarán para brindar información al resto del equipo sobre **qué** testearmos, **dónde** enfocamos nuestros esfuerzos y **por qué** creemos que nuestras pruebas podrían ser consideradas como *suficientemente buenas (o no)*.

8. Defectos

Tal como su nombre lo adelanta, aquí vamos a registrar los identificadores de los defectos que hayamos creado en el sistema de gestión de defectos que manejemos. En caso de no contar con dicha herramienta, se sugiere que podamos crear aquí el

reporte del defecto tan completo como nos sea posible, para que sea más simple su reproducción.

9. Inconvenientes

Dentro de los componentes que forman parte de una sesión, este último tiene por cometido el registrar todo inconveniente o impedimento que haya surgido y haya quitado tiempo a la sesión y la concreción de su objetivo. Habitualmente, los problemas o inconvenientes que se registran pueden consistir en potenciales defectos, problemas con los servidores, entre otros.

Ejemplo de una sesión

Ahora que vimos una breve explicación de cada uno de los componentes de una sesión, veamos a todos juntos en una sesión de ejemplo con un sistema bajo pruebas imaginario, al que llamaremos *FastPicker*.

1) OBJETIVO

Explorar una decisión creada en la nueva versión de *FastPicker*, revisando que el usuario final cuenta con guía clara a través de las opciones disponibles y los criterios de selección implementados en la versión 1.2.

2) ÁREAS

- Sistema Operativo | Windows 7, 32 y 64 bits
- Build | 1.3
- *FastPicker* | Selección Ágil
- *FastPicker* | Generador de Reportes
- Estrategia | Exploración y Análisis
- Oráculo | *FastPicker* versión 1 para *Desktops*

3) INICIO

DD/MM/AAAA - HH:MM am/pm

4) TESTER

Justin Case

5) ESTRUCTURA DE DIVISIÓN DE TAREAS

- DURACIÓN: Larga
- DISEÑO Y EJECUCIÓN DE PRUEBAS: 70%
- INVESTIGACIÓN Y REPORTE DE DEFECTOS: 15%
- ARMADO DE LA SESIÓN: 15%
- OBJETIVO vs. OPORTUNIDAD: 85 / 15

6) ARCHIVOS DE DATOS

- Documento_de_Especificaciones.pdf
- Documento_de_Usuario.Administrador.pdf
- FastPicker_Datos.sql

7) NOTAS DE PRUEBAS

[Prueba #1] <Texto con la descripción de la prueba.>

[Prueba #2] <Texto con la descripción de la prueba.>

[Prueba #3] <Texto con la descripción de la prueba.>

8) DEFECTOS

ID #12345: ¡Hola! Soy un *bug*, y mi nombre es...

9) INCONVENIENTES

[#1] Desde las 9am y hasta las 9.40am el servidor de pruebas no estuvo disponible, debido a inconvenientes con el servidor *web*. La sesión se pausó y se retomaron las actividades a partir de las 9.45am.

[#2] Durante la ejecución de la Prueba #2 durante la sesión, el servidor arrojó errores de tipo HTTP-500. Dichos errores fueron reportados a infraestructura y desarrollo, para lo que se creó el ticket con ID #7563.

Métricas de una sesión

Hasta el momento, hemos visto cómo un equipo de *testing* puede usar el recurso de *Testing* Basado en Sesiones. A continuación, veremos cuáles son los puntos que se toman en consideración para obtener métricas.

Para las mediciones, las métricas en las sesiones son extraídas a partir de:

- Cantidad de sesiones que se hayan completado.
- Cantidad de defectos y problemas que se hayan encontrado.
- Áreas funcionales que hayan sido cubiertas.
- Porcentaje de tiempo que se haya invertido en:
 - o El **armado de la sesión**.
 - o **Testing**.
 - o La **investigación y reporte de defectos**.

En la siguiente tabla, vemos un ejemplo de un reporte clásico, que contiene dos sesiones clásicas basadas en el modelo creado por James y Jonathan Bach²⁹. Ese modelo puede generarse a partir del uso de una herramienta gratuita disponible bajo el nombre **CLR-Sessions**³⁰.

Sesión	Fecha	Hora	Dur.	Ob.	Op.	Test	Def.	Setup	#Bugs	#Iss.	#T
et-jsb-010416-a	4/16/14	9:30 am	1h	1	0	0.8	0	0.2	0	3	1
et-jsb-010416-b	4/16/14	1:00 pm	2h	2	0	2	0	0	0	6	2

Ejemplo de reporte de sesiones

En esta tabla podemos ver la siguiente información:

- **Sesión:** Se incluye el título del reporte de la sesión.
- **Fecha, hora:** Fecha y hora en la que se dio inicio con la sesión en cuestión.
- **Dur. = Duración:** Tipo de sesión y su extensión de tiempo, como por ejemplo, indicando si fue corta, media o larga tal como se explicó antes.
- **Ob. = Objetivo:** En esta columna se incluye la cantidad total de trabajo dedicado al **objetivo** durante cada sesión. Los valores se expresan en horas que se calculan en base al porcentaje ingresado en el reporte de la sesión y en base a la duración de esta.
- **Op. = Oportunidad:** Cantidad total de tiempo dedicado a trabajo de *testing* por fuera del **objetivo** de la sesión. Se expresa en horas tal como el

²⁹ SBTM Session report checklist:

<<https://www.satisfice.com/download/sbtm-session-report-checklist>>

³⁰ Acceso a CLR-Sessions: <<https://github.com/aquaman/clr-sessions/archive/master.zip>>

objetivo. En el ejemplo se destinó todo el tiempo de ambas sesiones al objetivo.

- **Test:** Cantidad de tiempo dedicado de forma exclusiva al trabajo durante la sesión, testeando en base al **objetivo**. Cuanto mayor sea este porcentaje, entonces, mayor el tiempo que se dedicó por parte de los *testers* a trabajar en el **objetivo**.
- **Def. = Defectos:** Tiempo dedicado de la sesión a trabajar sobre la investigación y reporte de defectos. **A tener en cuenta:** este tipo de trabajo se reporta en la sesión si y solo si significa una interrupción al *testing* y se dedica a investigar un defecto y reportarlo.
- **Setup = Armado:** Cantidad de tiempo en el que se registran las interrupciones al trabajo de los *testers*. Es cualquier tipo de actividad que no sea específicamente **testing** o **reporte e investigación de defectos**. Típicamente, en esta categoría se incluye el buscar información, armar la sesión como tal, preparar configuraciones, equipamiento o *software* para poder trabajar.
- **# Bugs = Cantidad de defectos:** Cantidad total de defectos o *bugs* reportados en el trabajo de *testing* durante la sesión.
- **# Iss. = Issues = Problemas:** Cantidad total de problemas que se hayan encontrado durante la ejecución de las sesiones, relacionados a las **áreas** de cobertura. Los problemas pueden ser desde inconvenientes con los servidores de prueba, problemas con la infraestructura edilicia donde están estos servidores, dudas sobre el producto bajo prueba que se hayan escalado y no cuenten con respuesta, potenciales defectos que sin la consulta con un oráculo podrían ser falsos negativos, entre otros.
- **# T = Tester:** Cantidad de *testers* que van a estar dedicados trabajando sobre una sesión.

Comentarios finales del capítulo

Y es así que llegamos por el momento a una primera conclusión sobre el *Testing Exploratorio*, en la que podemos decir que es útil para:

- Proporcionar información y resultados de forma rápida.
- Obtener nueva información y conocimiento a lo largo de nuestro trabajo de *testing*.
- Revelar nuevos tipos de defectos y problemas.
- Mejorar las habilidades del *tester* y equipo de *testing* en general, además del conocimiento sobre la lógica del negocio.

Por último, notamos que el modo de registrar y guardar información en las sesiones es una tarea muy importante tanto para el *tester*, equipo de *testing*, como para el resto de los actores que forman parte del equipo.

La automatización del *testing* es *testing* asistido por computadoras.

Cem Kaner

AUTOMATIZACIÓN DE PRUEBAS FUNCIONALES

Se suele decir: “*Automating chaos just gives faster chaos*” y no solo *faster*, sino que (parafraseando una canción de *Daft Punk*, 2001) “*harder, faster, stronger...*” caos. En este capítulo, la idea es hablar de automatización de pruebas funcionales en general y mostrar los beneficios que aporta, de la manera más objetiva posible. Claro está que, si no se automatiza con criterio, entonces, no se obtendrá beneficio alguno. Este capítulo no es un manual de usuario de ninguna herramienta ni tampoco trata de convencer de que la automatización es la “varita mágica” que va a lograr que nuestras pruebas sean mucho mejores. Dicho esto, estamos convencidos de que debería considerarse al definir la estrategia de pruebas de cualquier tipo de sistema para analizar si mejora la eficiencia (y a veces efectividad) del *testing*.

Introducción a las pruebas automatizadas

Para hablar de *Testing* Automatizado, hablemos primero de pruebas de regresión. Si bien no es para lo único que se utiliza la automatización de pruebas, tal vez sea la situación más común en la que se usa.

Test de regresión

Las pruebas de regresión son un subconjunto de las pruebas planificadas que se seleccionan para ejecutar periódicamente, por ejemplo, ante cada nueva liberación del producto. Su objetivo es verificar que los cambios introducidos en la nueva liberación no afecten negativamente el comportamiento de los flujos ya conocidos, o en otras palabras, verificar que el producto no haya sufrido **regresiones**.

¿Por qué se llaman *pruebas de regresión*?

En un principio, cualquiera puede pensar que se refiere a regresar, a ejecutar las mismas pruebas, ya que se trata un poco de eso. Pero en realidad el concepto está asociado a verificar que no haya regresiones en lo que se está probando. ¡Queremos evitar esas regresiones! Y, por eso, se realizan estas pruebas.

Tampoco es válido pensar que las pruebas de regresión se limitan a verificar que se hayan arreglado los *bugs* que se habían reportado, pues es igual de importante ver que lo que antes estaba bien ahora siga funcionando.

Generalmente, cuando se diseñan las pruebas para determinadas funcionalidades, ya se definen cuáles son las pruebas que serán consideradas dentro del set de pruebas de regresión, o sea, las que serán ejecutadas ante cada nueva liberación del producto o en cada ciclo de desarrollo. Ejecutar las pruebas de regresión consistirá en ejecutar nuevamente las pruebas diseñadas con anterioridad.

Hay quienes dicen que, al contar con una listita con los pasos a seguir y las cosas a observar, no se está haciendo *testing*, sino un simple chequeo. En un par de artículos, James Bach y Michael Bolton comentan las diferencias entre el *testing* y el chequeo (*checking*). El *testing* es algo para lo que uno necesita creatividad, atención, busca caminos nuevos, piensa “¿de qué otra forma se puede romper?”. Al chequear, simplemente nos dejamos llevar por lo que alguien ya pensó antes, por esa ya mencionada lista de pasos.

Un problema que surge con esto es que las pruebas de regresión, viéndolas así, son bastante aburridas. El aburrimiento genera distracción. La distracción provoca errores. El *Testing* de Regresión está atado al error humano. ¡Es aburrido volver a revisar otra vez lo mismo! Eso hace que uno preste menos atención e incluso puede llegar a darse la situación de que desee que algo funcione e inconscientemente confunda lo que ve para dar un resultado positivo.

¡Ojo! ¡No estamos diciendo que el *testing* sea aburrido! A nosotros nos encanta. Estamos diciendo que las cosas rutinarias son aburridas y, por ende, propensas al error. Además, los informáticos al menos tenemos esa costumbre de ver las cosas que son automatizables y pensar cómo podríamos programar algo para no tener que hacer esta tarea a mano. Entonces es cuando podemos introducir *testing* automatizado, ya que los robots no se aburren. Veamos algún ejemplo para aclarar la idea de cuándo algo es automatizable o qué es probable que convenga automatizar:

- Necesitamos repetir **una serie de pasos una cantidad considerable de veces**, por ejemplo, revisar que en las 1000 páginas de un documento los títulos estén en mayúsculas.
- Necesitamos repetir **una serie de pasos de manera frecuente**, por ejemplo, todos los días crear un nuevo archivo que contenga la fecha, hora y cantidad de *stock* de cada producto de la base de datos.
- Necesitamos repetir **una misma acción con diferentes datos**, por ejemplo, obtener el precio y agregarle el impuesto correspondiente, para cada uno de los productos de la base de datos (y cada uno con un valor diferente).

- Necesitamos repetir **una serie de pasos en distintas plataformas**, por ejemplo, verificar que una *web* muestre su logo del tamaño adecuado en cada uno de los 10 celulares y *tablets* más populares del momento.

El **Testing Automatizado** consiste en que una máquina logre ejecutar los casos de prueba en forma automática leyendo la especificación de alguna forma (la lista de acciones), que pueden ser *scripts* en un lenguaje genérico o propio de una herramienta, a partir de planillas de cálculo, modelos, etc. Por ejemplo, Selenium³¹ (de las más populares herramientas *open source* para automatización de pruebas de aplicaciones *web*) tiene un lenguaje llamado *Selenese*, que brinda un formato para cada comando (acción) que puede ejecutar y, de esa forma, un *script* es una serie de comandos que respetan esta sintaxis. La herramienta, además, permite exportar directamente a una prueba JUnit³² en Java y otros entornos de ejecuciones de pruebas.

A continuación, compartimos algunas ideas tomadas de charlas con varias personas que desempeñan distintos roles dentro del proceso de desarrollo, como para entender mejor qué es lo que se busca hacer al automatizar este tipo de pruebas.

Desarrollador:

“Quiero hacer cambios sobre la aplicación, pero me da miedo romper otras cosas. Ahora, testearlas todas de nuevo me da mucho trabajo. Ejecutar pruebas automatizadas me da un poco de tranquilidad de que, con los cambios que hago, las cosas que tengo automatizadas siguen funcionando”.

Tester:

“Mientras automatizo, voy viendo que la aplicación funciona como es deseado y luego sé que lo que tengo automatizado ya está probado, eso me da lugar a dedicar mi tiempo a otras pruebas, con lo cual puedo garantizar más calidad del producto”.

Desarrollador:

³¹ Selenium: herramienta de pruebas web automatizadas. <<http://seleniumhq.org>>

³² JUnit: herramienta de pruebas unitarias automatizadas. <<http://junit.org>>

“Hay veces que no innovo por miedo a romper, principalmente cuando tengo sistemas muy grandes, donde cambios en un módulo pueden afectar a muchas funcionalidades”.

Usuario:

“Cuando me dan una nueva versión de la aplicación, no hay nada peor que encontrar que lo que ya andaba dejó de funcionar. Si el error es sobre algo nuevo, es entendible, pero que ocurra sobre algo que ya se suponía que andaba cuesta más. Las pruebas de regresión me pueden ayudar a detectar estas cosas a tiempo, antes de que el sistema esté en producción”.

Claro que luego surge otro tipo de cuestiones, como si hay que automatizar las pruebas de regresión sí o sí o se pueden hacer total o parcialmente de manera manual. Para llegar a un buen fin con estas decisiones, esperamos brindarles más herramientas y argumentos a lo largo de este capítulo.

Retorno de la inversión

“El costo de un único defecto, en la mayoría de las organizaciones, puede cubrir el precio de una o más licencias de herramientas” (W. Rice, 2003).

¿Costo o inversión? Generalmente se dice que el *Testing* Automatizado permite ampliar la cobertura y alcance de las pruebas, reducir costos, focalizar las pruebas manuales en lo que realmente interesa, encontrar defectos más temprano, etc., etc., etc., y de esa forma reducir costos y riesgo. Todo esto parece ser siempre la misma estrategia de *marketing* que plantean los vendedores de herramientas, sin justificación suficiente. No es que nada de eso sea cierto, pero hay que analizarlo y fundamentar mejor.

Es interesante apuntar a lo que plantea Paul Grossman en un artículo titulado *Automated testing ROI: fact or fiction?*. Allí se habla de los beneficios de la automatización de pruebas en términos financieros, más que nada para dar más argumentos a quien quiera visualizar o mostrar el valor de este tipo de inversiones.

Lo importante es ver que no se trata de un gasto, sino de una inversión para ganar no solo en calidad, sino en ahorro de dinero. Porque también podríamos

preguntarnos si la calidad es un costo, si al brindar productos de calidad tendrán que ser más caros, ya que cuesta más construirlos (en esfuerzo y dinero), o si también producir con calidad nos ayuda a ahorrar en otros costos asociados a la falta de calidad.

Para las pruebas, aunque sean “automáticas” y “ejecutadas por una máquina”, vamos a necesitar gente capacitada. Como bien dice Cem Kaner (2001), en un capítulo del libro *Lessons learned in software testing* dedicado a automatización, una herramienta no enseña a los *testers* cómo testear; si el *testing* es confuso, las herramientas refuerzan esa confusión. Recomienda corregir los procesos de *testing* antes de automatizar.

Además, no es la idea reducir el personal dedicado a *testing*, pues el test manual aún se necesita y las pruebas automatizadas requieren cierto esfuerzo para su construcción y mantenimiento. No ahorraremos dinero en personal, entonces ¿dónde están los beneficios financieros?

Lo más importante es que al automatizar podemos expandir dramáticamente el alcance de las pruebas con el mismo costo (con la misma inversión), y eso es algo que financieramente da ventajas claras: conservar los costos y aumentar el beneficio.

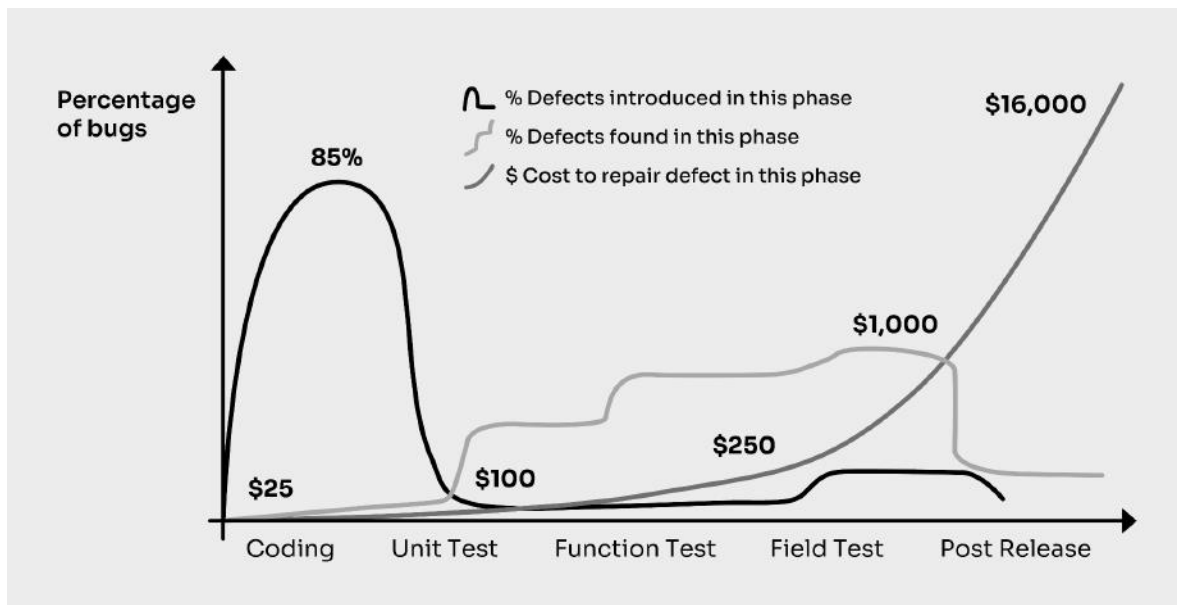
Veamos esto con un ejemplo hipotético que sea más entendible por alguien que decide sobre las finanzas de una empresa: La persona encargada de *testing* ejecuta manualmente pruebas 8 horas por día y se va a su casa. En ese momento, el *testing* se detiene. Con el test automatizado se puede estar ejecutando pruebas 16 horas más por día (en el mejor caso, claro...), bajo el mismo costo, lo cual reduce el costo de las horas de test. Paul Grossman en su libro hace un análisis financiero en base a costos por hora de las personas involucradas en el desarrollo y *testing* que es interesante para profundizar sobre el tema, pero en sus cálculos termina mostrando que el costo de probar con automatización es mucho menor que sin ella.

Claramente, estos siguen siendo argumentos de vendedores... Tal vez suceda en un mundo ideal, pero de todos modos, si apuntamos a esta utopía, lograremos mejoras

en nuestros beneficios (manteniendo los costos) y ampliando los horizontes (basándonos en el concepto de *utopía*³³ expresado por Eduardo Galeano en 2003).

Ahora, esto nos muestra claramente que hay mejoras en cuanto a las horas de ejecución de pruebas, pero seguramente nos agregue más costo, pues ¡encontraremos más errores! ¿Queremos eso? ¡Claro que **sí!**, y veremos ahora cómo encontrar más errores nos ahorra costos aunque nos da más trabajo (es algo obvio, pero hagamos algunas cuentas para visualizarlo mejor).

La siguiente figura muestra una gráfica bien conocida en ingeniería de *software* (Jones, Carpers, 1996), que representa el costo de corregir un defecto detectado según en la etapa en que se haya encontrado (diseño, codificación, desarrollo, integración, *testing*, producción). El estudio data de varios años, pero sirve para dar una idea relativa entre los distintos momentos del desarrollo.



Costo de resolución de *bugs*

Esto no está considerando siquiera los costos ocultos que hay, como la pérdida de imagen, confianza, e incluso el desgaste del equipo.

³³ “La utopía está en el horizonte. Camino dos pasos, ella se aleja dos pasos y el horizonte se corre diez pasos más allá. ¿Entonces para qué sirve la utopía? Para eso, sirve para caminar”.

Lo que se concluye de esto es algo ya muy conocido: cuanto antes se encuentren los *bugs*, mejor. Y, como vamos a tener *testers* que trabajen 24hs al día y ejecutando más rápido (los *scripts* automatizados!), es más probable que se encuentren más errores antes de pasar las versiones a beta *testing* o a producción. Es difícil estimar cuánto, pero por cada *bug* que se encuentre más temprano se ahorrará mucho dinero.

A modo de resumen, podemos decir que hay dos cosas a las que el test automatizado le agrega valor:

- *Business value*: Da valor al negocio mejorando la calidad (complementa al *Testing Manual*), evita problemas de operativa, pérdida de imagen ante clientes e incluso problemas legales.
- *IT value*: Mejora el trabajo del grupo de IT, ya que simplifica las tareas rutinarias, permite que con el mismo costo haga mucho más y mejor.

¿Cuándo se hacen visibles los resultados?

Tal vez uno piense que, si las pruebas encuentran un error, es ahí cuando se están encontrando beneficios, entonces se podrá medir la cantidad de *bugs* detectados por las pruebas automatizadas. En realidad, el beneficio se da desde el momento de modelar y especificar con un lenguaje formal las pruebas a realizar. Luego, la información que da la ejecución de las pruebas también aporta un gran valor.

No es solo útil el resultado de error detectado, sino el resultado de *OK* que está diciendo que las pruebas que ya automatizamos están cumpliendo lo que verifican. Un artículo de *Methods and tools* dice que **se encuentran gran cantidad de *bugs* al momento de automatizar los *test cases***. Al automatizar, se debe explorar la funcionalidad, probar con distintos datos, etc. Generalmente, lleva un tiempo en el que se está “dándole vueltas” a la funcionalidad a automatizar. Luego, para probar que la automatización haya quedado bien, se ejecuta con distintos datos, etc. En ese momento, ya estamos haciendo un trabajo de *testing* muy intenso.

¡Cuidado! Si automatizamos pruebas sobre un módulo y consideramos que con esas pruebas es suficiente, entonces ¿no lo probamos más? El riesgo es que las pruebas automatizadas no estén cubriendo todas las funcionalidades (paradoja del pesticida). Dependencia con la calidad de los test. Tal vez tengamos mil pruebas, y por eso creamos que tenemos un buen *testing*, pero esas pruebas quizá no verifican lo suficiente, son superficiales o todas similares.

El valor de automatizar no se ve en la cantidad de pruebas ni en la frecuencia con que ejecuto esas pruebas, sino por la información que proveen (que complementa la información que nos aporta una persona que ejecuta las pruebas en forma manual).

¿Por qué y para qué automatizar?

Pasemos a hablar ahora directamente de automatización de pruebas. Si nos basamos en una definición tradicional de automatización, podemos decir que se refiere a una tecnología que permite automatizar procesos manuales, lo que ocasiona varias ventajas asociadas:

- Mejora la calidad, pues hay menos errores humanos.
- Mejora la *performance* de producción, pues con las mismas personas se puede lograr mucho más trabajo a mayor velocidad y escala, y en ese sentido, mejora el rendimiento de las personas.

Esta definición de la automatización, tomada de la automatización industrial, aplica perfectamente a la automatización del *testing* (o del *checking*, si seguimos la diferenciación que comentamos que hacen Michael Bolton y James Bach).

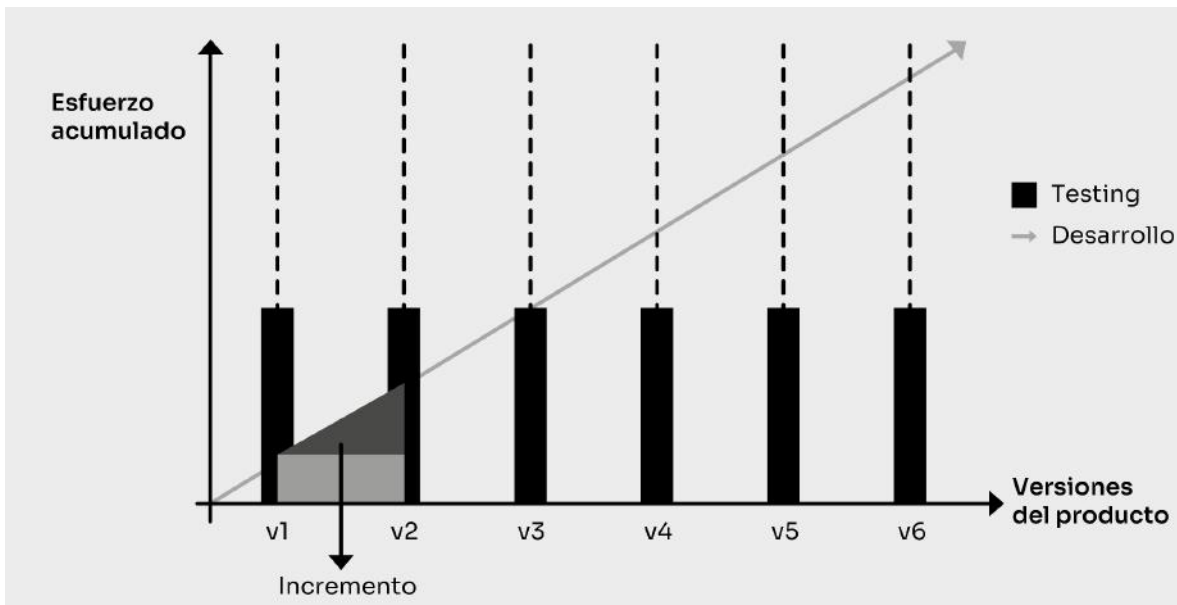
Si bien esta es una idea que venimos afirmando desde el comienzo del capítulo, queremos plantear una teoría a la que le llamamos *acumulatividad nula*³⁴. Para eso, veamos la gráfica en la siguiente figura, donde se muestra que las *features* crecen cada vez más a lo largo del tiempo (de una versión a otra), pero el test no

³⁴ Basada en algo que se plantea en el libro *Software test automation: effective use of test execution tools*, de Mark Fewster y Dorothy Graham, página 494.

<https://www.goodreads.com/book/show/637584.Software_Test_Automation>

(no conocemos ninguna empresa que a medida que desarrolle más funcionalidades contrate más *testers*).

Esto es un problema, pues significa que cada vez más se nos da la situación de tener que elegir qué testear y qué no, y dejamos muchas cosas sin probar. El hecho de que las *features* vayan creciendo con el tiempo implica que el esfuerzo en *testing* debería aumentar también proporcionalmente. De aquí surge el problema de no tener tiempo para automatizar, pues no hay siquiera tiempo para el test manual.



Representación de la acumulatividad nula

Se suele decir que lo más pesado (costoso) en *testing* es diseño y ejecución. Podríamos considerar que el diseño es acumulativo, pues vamos diseñando y registrando en planillas de cálculo o documentos. El problema es que la ejecución de pruebas no es acumulativa. Cada vez que liberamos una nueva versión del sistema es necesario (deseable, sería necesario) testear todas las funcionalidades acumuladas y no solo las del último incremento, pues tal vez funcionalidades que se implementaron en versiones anteriores cambien su comportamiento esperado por “culpa” de los últimos cambios.

Lo bueno es que **la automatización es acumulativa**. Es la única forma de hacer que el *testing* se haga constante (sin que se requieran más esfuerzos a medida pase el tiempo y crezca el *software* a probar).

El desafío es hacer *testing* en forma eficiente, de forma que nos rinda, que veamos los resultados, que nos aporte valor y que vaya acompañando la *acumulatividad* del desarrollo.

¡Ojo! Los casos de prueba tienen que ser fácilmente “mantenibles” si no, no se puede lograr esta acumulatividad en forma efectiva.

Un motivo muy importante (y muy mencionado) de por qué automatizar es que nos permite **aumentar la calidad del producto**. Por ejemplo, si automatizamos las pruebas de regresión y hacemos que los *testers* puedan dedicar más tiempo a comprobar el funcionamiento de las nuevas funcionalidades, entonces, logramos aumentar el cubrimiento del *testing* y de esa forma contribuimos a la calidad final. Ejecutar pruebas de regresión en forma manual hace que se pierda la motivación del *tester*, pues las primeras veces que se ejecutan las pruebas se tiene la motivación de encontrar errores, pero luego de ejecutar varias veces lo mismo se pierde la atención, se deja de verificar cosas por el hecho de que “siempre funcionaron, ¿por qué dejarían de funcionar ahora?”.

Por otra parte, al automatizar pruebas logramos (a mediano y largo plazo) reducir el tiempo insumido en las pruebas de una determinada funcionalidad, con lo cual podremos **disminuir el tiempo de salida al mercado**.

Ejecutar las pruebas automatizadas más rápido, y con mayor frecuencia, ya que es menos costoso que hacerlo en forma manual, nos **permite detectar antes los errores** introducidos sobre funcionalidades ya existentes del sistema.

Simplemente ejecutando las pruebas que tenemos preparadas podemos obtener un rápido *feedback* sobre el estado de calidad del sistema. Esto permite reducir no solo costos en el área de *testing*, sino también nos ayuda a **reducir costos de desarrollo**. Para el desarrollador, es más fácil corregir algo que “rompió” ayer que algo que “rompió” hace unos meses. En este sentido, es muy importante detectar rápidamente los errores.

La automatización brinda la capacidad de **testear en paralelo, en forma desatendida y en distintas plataformas**. Es posible así, con los mismos costos, verificar cómo funciona la aplicación en Internet Explorer y en Firefox, por ejemplo. Si esto se pretende hacer de forma manual, se duplicarían los costos, o sea, los mismos casos de prueba deberían ejecutarse en cada plataforma, pero en cambio automatizando las pruebas es simplemente indicar que se ejecuten sobre un entorno u otro.

¿Automatizar o no automatizar?

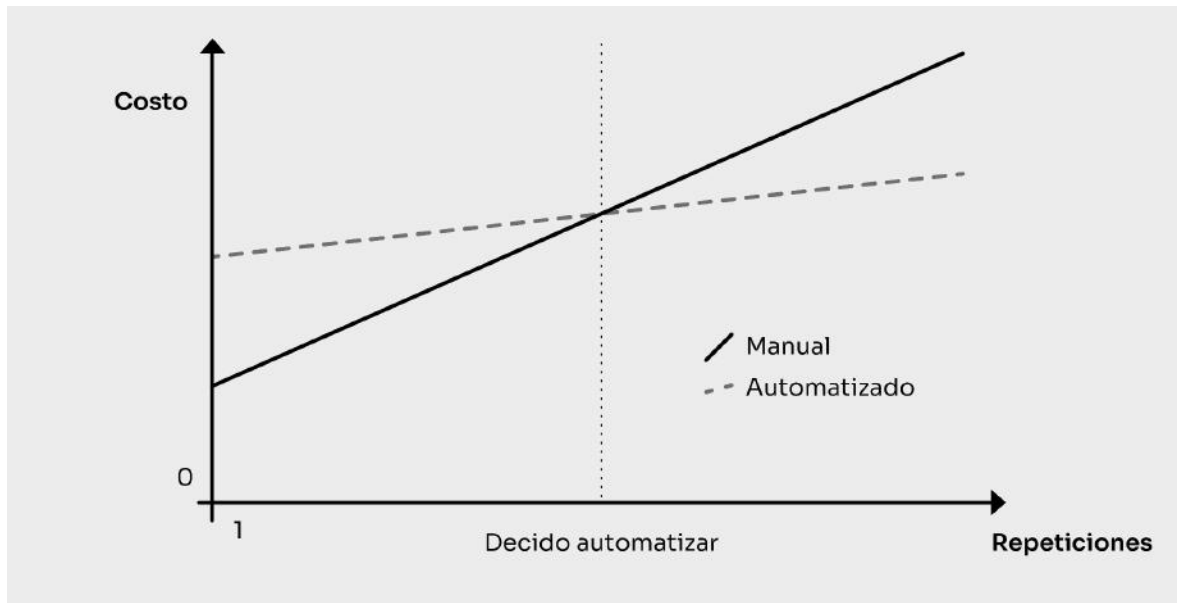
Como ya dijimos, luego de diseñar las pruebas, hay que ejecutarlas cada vez que haya cambios en el sistema, ante cada nueva liberación de una versión, por ejemplo. Esto es lo que se conoce como *test de regresión*, y si bien son bien conocidos sus beneficios, también es sabido que se requiere cierto esfuerzo para automatizar las pruebas y mantenerlas. Casi todas las herramientas de automatización brindan la posibilidad de “grabar” las pruebas y luego poder ejecutarlas, lo que se conoce como enfoque de *Record and Playback*. Esto generalmente sirve para pruebas simples y para aprender a usar la herramienta, pero cuando se quieren hacer pruebas más complejas es necesario conocer un poco más a fondo la forma en que trabaja la herramienta, manejar juegos de datos de pruebas, administrar los servidores de prueba, las bases de datos de pruebas, etc. Una vez que tenemos solucionado esto, podemos ejecutar las pruebas tantas veces como queramos con muy poco esfuerzo.

Las aplicaciones en las que más conviene usar *Testing* Automatizado son entonces las que en algún sentido tienen mucha repetitividad, ya que será necesario ejecutar muchas veces las pruebas (ya sea porque es un producto que tendrá muchas versiones, que se continúe con el mantenimiento haciendo *fixes* y parches o porque se debe probar en distintas plataformas).

Si se automatizan las pruebas de un ciclo de desarrollo, para el siguiente ciclo las pruebas automatizadas podrán volver a testear lo que ya se automatizó con bajo esfuerzo; permitirán que el equipo de pruebas aumente el tamaño del conjunto de pruebas y aumentarán el cubrimiento. Si no fuera así, terminaríamos teniendo

ciclos de pruebas más grandes que los ciclos de desarrollo (y cada vez más grandes) u optaríamos por dejar cosas sin probar, con los riesgos que eso implica.

La gráfica de la siguiente figura muestra el costo de las pruebas en función de uno de los factores más importantes, la cantidad de veces que vamos a repetir la ejecución de un caso de prueba.



Decisión de automatizar o no automatizar

El costo de una sola repetición es mayor para el automatizado, pero las pendientes son distintas. Aclaremos que la pendiente de las pruebas automatizadas no es 0, porque lleva un costo de mantenimiento. Hay un punto en el que las rectas se cruzan, y ese es el número de repeticiones que marca la inflexión. Si ejecutaremos el caso de prueba menos de esa cantidad de veces, es mejor no automatizar, si vamos a ejecutarlo más veces, entonces, es mejor automatizar.

La cantidad de veces está determinada por muchas cosas: por el número de versiones de la aplicación que vamos a probar, las distintas plataformas sobre las que la vamos a ejecutar o incluso los datos, pues, si un mismo caso de prueba lo tenemos que ejecutar muchas veces con distintos datos, eso también lo tendremos que tener en cuenta al decidir si automatizar o no.

Principios básicos de la automatización de pruebas

El que crea que una herramienta soluciona todos los problemas tiene un nuevo problema.

En general (y creo que más aún en el mundo IT) buscamos una herramienta que solucione **todos** nuestros problemas. El tema es que no es suficiente con tener una buena herramienta para realizar un buen trabajo.

Algunas reflexiones que se han hecho al respecto:

- “Automatizar el caos solo da caos más rápido” (del inglés: *Automating chaos just gives faster chaos*, por James Bach).
- “Si no conoces qué tests son los más importantes y cuáles son los más aplicables para la automatización, entonces, las herramientas solo te ayudarán a hacer un mal *testing* más rápido” (Fewster, Graham, 1999).

En esta sección comenzaremos a ver algunos elementos a tener en cuenta para que nuestras pruebas realmente nos den los beneficios que buscamos y que no fracasemos en el intento de automatizar nuestro test. Verán que el resto de esta sección tiene un orden casi cronológico, en el sentido de que comenzaremos con conceptos básicos y luego veremos las distintas actividades y formas de diseñar las pruebas en el mismo orden en que lo podrían hacer en sus proyectos de automatización de pruebas.

Automatización en distintos niveles

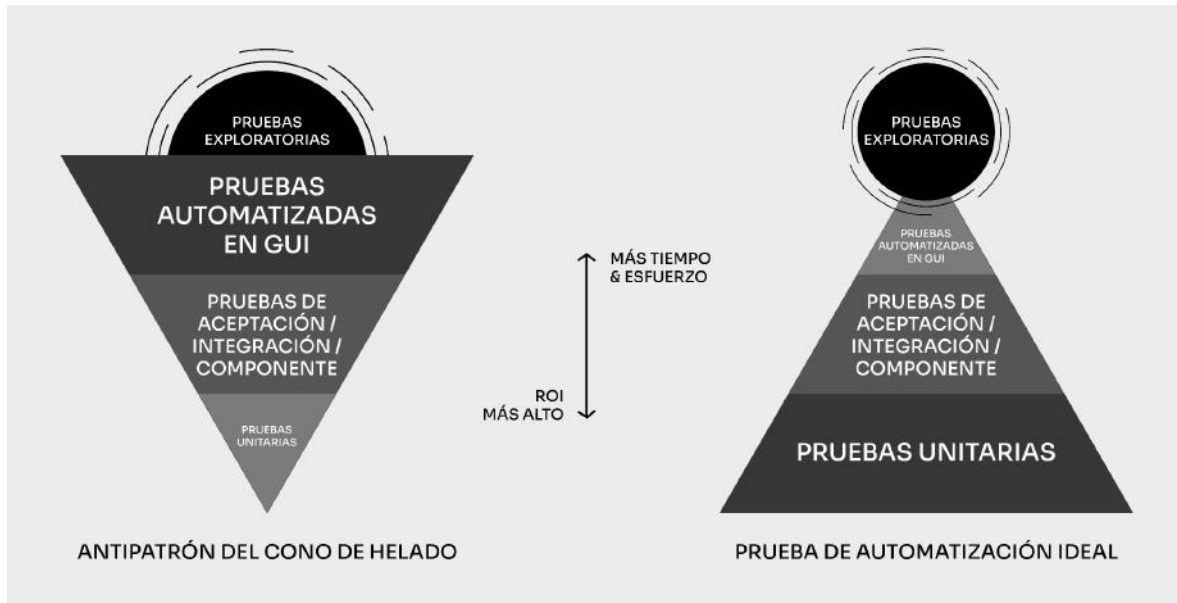
Generalmente se piensa en el *Testing* Automatizado a nivel de interfaz gráfica, pensando en un test que ejecuta similar a como lo haría un usuario, esto es, si hablamos de un sistema *web*, acceder a una URL y comenzar a presionar botones, seleccionar elementos, llenar campos con datos, revisar contenidos mostrando al

usuario, etc. Esta visión es un poco limitada, ya que hay otros niveles donde la automatización tiene muchísima relevancia. Estos otros niveles son las pruebas unitarias y las pruebas de API (del inglés *Application Programming Interface*, Interfaz de Programación de Aplicaciones). Sin entrar en demasiados detalles técnicos, aclaremos qué son estos niveles:

- **Pruebas unitarias:** son un tipo de prueba mediante la que se evalúa de manera individual y aislada cada componente o unidad de código, revisando que funcione correctamente lo que ayuda a identificar y corregir errores de forma temprana en el proceso de desarrollo. Estas pruebas son comúnmente llevadas a cabo por desarrollo.
- **Pruebas de API:** una API es un punto de acceso que permite que diferentes aplicaciones se comuniquen entre sí, siguiendo determinados protocolos y especificaciones. Entonces, una API actúa como un puente que permite que las aplicaciones intercambien información y funcionalidades. Es un punto de acceso también para probar esas funcionalidades y son especialmente útiles para automatizar estas pruebas.
- **Pruebas de interfaz de usuario:** también son conocidas como pruebas *End to End*, haciendo referencia a que al ejercitar el sistema desde su interfaz de usuario se estarán ejercitando todas sus capas de punta a punta, incluyendo los componentes visuales, el código que incluye la lógica de negocio del sistema y los datos en las distintas fuentes (bases de datos, archivos).

Relacionado a los niveles de pruebas automatizadas, existe un modelo muy popular conocido como la *pirámide de pruebas*, que fue propuesto por Mike Cohn (2009). Básicamente, el modelo plantea que es mejor tener una base firme de pruebas unitarias (más rápidas, *feedback* más temprano, más cerca del desarrollador, más baratas de hacer y mantener) y tener una cantidad reducida de pruebas a nivel de interfaz gráfica, ya que son lentas y poco confiables en comparación al resto. Las pruebas de API quedan en el medio y típicamente son llevadas a cabo tanto por *testing* como por desarrollo. En algunas representaciones del modelo de la pirámide se suele representar el *Testing Manual* como algo que va por encima de la pirámide.

En la siguiente figura, se puede ver el modelo de la pirámide de pruebas automatizadas, incluyendo el círculo que representa las pruebas manuales, así como lo que se conoce como el *anti-patrón* del helado. En el modelo del cono de helado, lamentablemente, se representa la práctica más habitual que mencionamos al iniciar esta sección, ya que al pensar el *Testing* Automatizado se lo concibe como automatizar solo a nivel de interfaz gráfica.



Para profundizar más en el modelo de la pirámide, les recomendamos seguir con el artículo de Martin Fowler (2018), que trata el tema desde un punto de vista muy práctico y pragmático.

Paradigmas de automatización

Existen diversos enfoques de automatización y para cada contexto nos serán más útiles algunos que otros. Es bueno tenerlos presentes también para el momento de seleccionar la estrategia de pruebas e incluso para seleccionar las herramientas más adecuadas según nuestro sistema bajo pruebas. Veamos tres enfoques que, según nuestra experiencia, son los más comunes y a los que sacaremos más provecho

(siempre pensando más que nada en automatizar a nivel de interfaz de usuario o API).

Scripting

Este es de los enfoques más comunes de pruebas automatizadas. Generalmente, las herramientas brindan un lenguaje en el que se pueden especificar los casos de prueba como una secuencia de comandos que logran ejecutar acciones sobre el sistema bajo pruebas.

Estos lenguajes pueden ser específicos de las herramientas o pueden ser un componente de programación para un lenguaje de propósito general, como lo es JUnit para Java³⁵.

De acuerdo al nivel que ejecutan las pruebas en la herramienta de automatización, será el tipo de comandos proporcionados por la herramienta. Hay herramientas que trabajan a nivel de interfaz gráfica, entonces, tendremos comandos que permitan ejecutar acciones como clics o ingreso de datos en campos de una pantalla. Otras trabajan a nivel de protocolos de comunicación, entonces tendremos acciones relacionadas con ellos, como puede ser la herramienta HttpUnit³⁶ a nivel de http³⁷, la cual nos da la posibilidad por ejemplo de ejecutar comandos GET y POST que son específicos del protocolo.

A continuación tenemos una pequeña porción de código, que si bien no es completamente real, pretende ayudar a quienes lean a entender el mecanismo de automatización con código. Veamos parte por parte:

- Por un lado, se especifican aspectos exigidos por la herramienta JUnit para declarar que esa porción de código corresponde a un test, al cual lo vamos a identificar con su nombre, *testCase01*.
- En la línea 6, se puede apreciar que se invoca una funcionalidad del sistema que recibe un parámetro (*testData*). Se hace a través de lo que se conoce como *objeto* llamado *systemUnderTest*. El resultado de esta invocación se almacena en lo que se conoce como una variable llamada *actual*.

³⁵ Java es un lenguaje muy popular, uno de los tres más usados a la fecha.

³⁶ Información sobre HttpUnit: <<http://httpunit.sourceforge.net>>

³⁷ Http: protocolo de comunicación sobre el cual funcionan los sitios *web*, por ejemplo.

- Luego de la ejecución, se hace una verificación con un método provisto por JUnit llamado *assertEquals*. Esto, básicamente, compara las variables *expected* (previamente definida) y actual (la que retornó la ejecución de nuestra prueba) y, en caso de que falle, mostrará el mensaje que aparece entre comillas como primer parámetro.

```
1 public class TestSuite {
2     @Test
3     public void testCase01() {
4         String testData = "input";
5         String expected = "expectedOutput";
6         String actual = systemUnderTest.execute(testData)
7         assertEquals("failure", expected, actual);
8     }
```

Ejemplo de prueba con JUnit

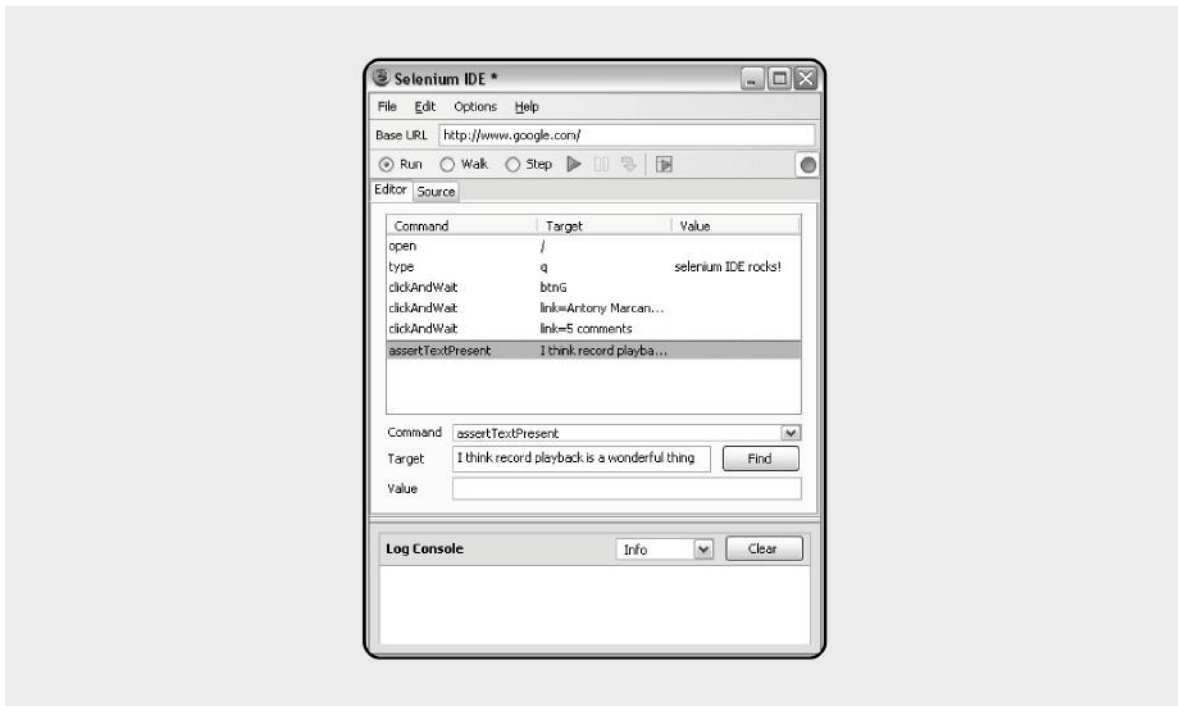
La forma en la que típicamente se hace esto es pasar por parámetro los datos de prueba y usando Selenium (o alguna herramienta que permita interactuar con la interfaz del sistema) se cargarán esos parámetros en los campos de entrada correspondiente.

Veamos el mismo procedimiento de prueba, pero para un ejemplo donde se está probando el acceso a una página *web*. Invitamos a ustedes a revisar el código y, en base a lo explicado previamente, intentar descifrar cómo funciona el test.

```
1 public class TestSuite {
2     @Test
3     public void testCaseLogin01() {
4         String testdata_username = "federico.toledo";
4         String testdata_password = "pass123$";
5         String expected = "Bienvenido Federico!";
6         String actual = loginPage.execute(testdata_username,
                                           testdata_password)
7         assertEquals("Fallo al ingresar", expected, actual);
8     }
```

No es tan complicado de leer y entender, ¿verdad?

En la siguiente figura se observa un ejemplo de una prueba automatizada con Selenium (si bien la captura tiene varios años, el concepto se mantiene). En ella se puede ver que primero se cargan valores en dos *inputs* con el comando “*type*” y luego se hace clic en el botón para enviar el formulario. Nuevamente los invitamos a revisar el *script* de prueba automatizada para intentar entenderlo.



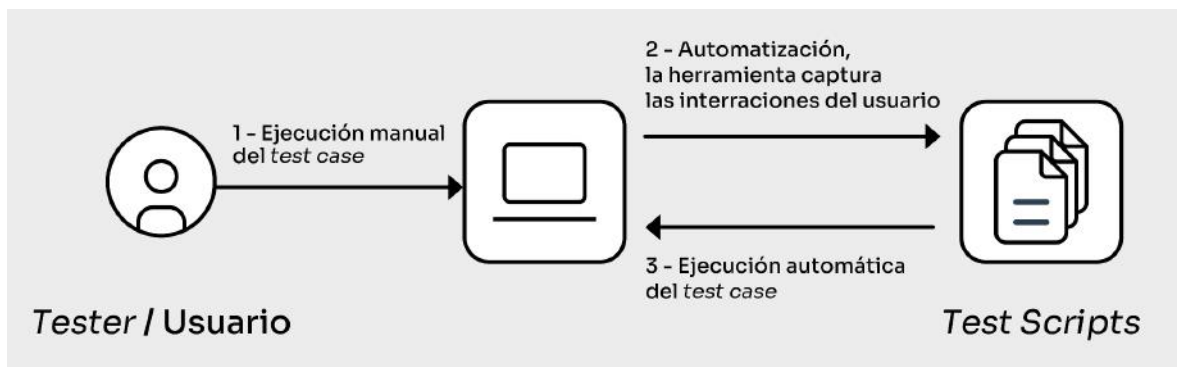
Ejemplo de prueba con Selenium

Para preparar pruebas automatizadas siguiendo el enfoque de *scripting* es necesario programar los *scripts*. Como seguramente quedó claro en los ejemplos, para poder hacer esto deberemos conocer el lenguaje de la herramienta y los distintos elementos con los que interactuamos del sistema bajo pruebas, como pueden ser los botones en una página *web*, los métodos de la lógica que queremos ejecutar o los parámetros a enviar en un pedido GET de un sistema *web*. Es una tarea muy entretenida, incluso lúdica, con la que se va ganando experiencia y agilidad con la práctica.

Record and Playback

Como la tarea de programar los *scripts* generalmente es muy costosa, este paradigma de “grabar y ejecutar” nos va a permitir generar (al menos la base) de los *scripts* en forma sencilla.

La idea es que la herramienta sea capaz de capturar las acciones del usuario (*record*) sobre el sistema que estamos probando y pueda volcarlo luego a un *script* que sea reproducible (*playback*). La siguiente figura representa en tres pasos este proceso, donde el usuario ejecuta manualmente sobre el sistema bajo pruebas, en ese momento la herramienta captura las acciones y genera un *script* que luego puede ser ejecutado sobre el mismo sistema.



Enfoque de *record & playback*

Sin este tipo de funcionalidad, sería necesario escribir los casos de prueba manualmente, y para eso, como decíamos, se necesita conocimiento interno de la aplicación, así como también en el lenguaje de *scripting* de la herramienta utilizada. Si la persona que realiza estas tareas no cuenta con conocimientos en programación puede ser muy complejo y, por eso, es deseable contar con esta funcionalidad.

Los *scripts* que se generan a partir de la captura de las acciones de usuario generalmente se deben modificar, para lo que es necesario conocer el lenguaje y los elementos del sistema bajo pruebas, pero será mucho más fácil editar un *script* generado que programarlo completamente de cero. Entre las modificaciones que pueden ser necesarias o útiles podemos incluir la parametrización de las pruebas, para que el *script* tome distintos datos de prueba (siguiendo el enfoque

Data-Driven Testing), agregar cierta lógica a los *scripts* como estructuras *if-then-else*, en caso de que sea necesario seguir distintos flujos o estructuras de bucle, en caso de que sea interesante ejecutar ciertas cosas repetidas veces.

Los *scripts*, de esta forma, los podemos grabar a partir de nuestra ejecución de un caso de prueba sobre la aplicación. Esto lo veremos más adelante, pero para automatizar, primero es necesario que diseñemos las pruebas y luego las grabemos con la herramienta. Teniendo esto en mente, podríamos decir que la tarea que pasa a ser de las más complejas y costosas es la del diseño de pruebas.

Model Based Testing / Model Driven Testing

El siguiente nivel de automatismo implica automatizar no solo la ejecución de las pruebas, sino también su diseño. Para esto, el planteo es utilizar algo que se conoce como *enfoque dirigido por modelos*, los cuales pueden provenir de dos fuentes distintas:

- Modelos diseñados para *testing*
- Modelos de desarrollo

Por un lado, este enfoque puede contar con que el *tester* generará de alguna forma un modelo específico para la generación de pruebas, como por ejemplo, una máquina de estados o cualquier otro tipo de modelo con información de cómo debería comportarse el sistema bajo pruebas. Por otro lado, podrían aprovecharse ciertos artefactos de desarrollo o de la propia aplicación para, a partir de esa información, generar pruebas. Esto pueden ser por ejemplo los diagramas UML utilizados en etapas de diseño, casos de uso, *user stories*, el esquema de la base de datos o, si estamos hablando de un sistema desarrollado con GeneXus³⁸, lo que se conoce como *la base de conocimientos* (comúnmente llamada KB, por su sigla en inglés).

El resultado obtenido dependerá de cada herramienta, pero generalmente serán casos de prueba especificados en cierto lenguaje, datos de prueba o *scripts* de

³⁸ GeneXus: <<http://www.genexus.com>>

pruebas automáticas para poder ejecutar directamente los casos de prueba generados.

De esta forma, las pruebas están basadas en una abstracción de la realidad mediante un modelo. Así se evita tener que lidiar con los problemas técnicos, y es posible concentrarse solo en el modelado del problema, lo que permite que las pruebas sean más fáciles de entender y mantener.

En este libro seguiremos hablando principalmente de automatización con *scripting*, apoyándonos en herramientas del tipo *Record and Playback*, que permitan parametrizar sus acciones para poder seguir un enfoque de *Data-Driven Testing*. También daremos sugerencias relacionadas al diseño de las pruebas y de distintos aspectos sobre la automatización, considerando que el diseño se hará manualmente, no necesariamente con herramientas que usen técnicas basadas en modelos. Para profundizar sobre herramientas que usan técnicas de *Model-Based Testing*, se puede, por ejemplo, ver la sección de Máquinas de Estado de este libro y las herramientas presentadas ahí.

Low Code / Scriptless

En los últimos años el término *Low Code* (que podríamos traducir como *de bajo código*, pero siempre se usa en inglés) tomó mucha fuerza. El mismo concepto se encuentra también como *no-code*, *scriptless* o *codeless*.

Básicamente, *Low Code* se refiere a un enfoque alternativo al desarrollo de *software* tradicional. En lugar de definir el comportamiento esperado del sistema en código, tiende a ser más visual y usar poco o ningún código en absoluto. Estamos aumentando el nivel de abstracción del lenguaje que usamos para expresar a las máquinas qué hacer. Comenzamos programando circuitos, cadenas de 0 y 1, instrucciones de ensamblador, y luego diferentes lenguajes que proporcionan capas de abstracción cada vez mayores. Cada capa elimina algo de complejidad, cerrando la brecha entre el humano y la máquina. La contraparte es que la abstracción también tiene un costo, típicamente asociado con el rendimiento o la flexibilidad, pero normalmente tiene sentido pagar ese costo en términos de productividad y resultados.

Podríamos decir que el *Low Code* es otra capa de abstracción en la parte superior, y también estamos viendo cómo podemos beneficiarnos de este enfoque para la automatización de pruebas. Las plataformas de *Low Code* para la automatización de pruebas tienen como objetivo simplificar la automatización de pruebas con funcionalidades que no requieren que el usuario escriba casi ningún código. Tendremos un *recorder* que nos permitirá crear casos de prueba fácilmente, y editarlos con una interfaz sencilla, sin necesidad de habilidades de codificación.

En los últimos años, han surgido muchas soluciones de *Low Code* para la automatización de pruebas, incluso hicimos un análisis de todas las soluciones existentes al 2021 y lo publicamos en nuestro blog.

Selección de la herramienta

Como venimos viendo, existen muchas formas de automatizar, diferentes paradigmas y niveles, y esto hace que haya muchas herramientas. ¿Cómo elegir la que más nos conviene? Esta decisión puede ser una de las más complejas, pero se deberán tener en cuenta, básicamente, las características del proyecto, los objetivos, la estrategia de calidad, el presupuesto disponible, etc., y nos gustaría destacar que vemos necesario considerar características del equipo. En este sentido, se incluye cuáles son sus conocimientos y habilidades, y cuáles son sus preferencias de tecnologías y metodologías de trabajo. Por ejemplo, si se trabaja con metodologías ágiles que promueven la colaboración entre *testing* y desarrollo, evitando la formación de silos o separaciones, y el equipo de desarrollo programa con Java, buscaremos una herramienta de automatización basada en Java para facilitar la colaboración en torno a la tarea de automatización.

Existen diversas herramientas ***open-source***, **comerciales**, y **customizadas**, que varían en sus limitaciones y posibilidades, por lo que, para seleccionar la más adecuada, hay que tener claros los requerimientos con los que se debe contar para avanzar a evaluar el costo-beneficio de su uso.

Se hace importante destacar que no hay mejores herramientas para todos los casos. Sí podemos elegir entre aquellas que nos ofrezcan mayor flexibilidad, aunque

siempre va a depender de la aplicación que se tenga bajo prueba y los criterios por los cuales se tomen las decisiones.

Diseño de pruebas según objetivos

Como todo en la vida, en la automatización de pruebas también hay que tener presente un objetivo. Pensar para qué queremos que sean útiles las pruebas automatizadas y luego actuar en consecuencia. O sea, de acuerdo a cuáles sean los objetivos que planteemos para la automatización, tendré que tomar ciertas decisiones para seguir por un camino u otro, para seleccionar unos casos de prueba en lugar de otros o para diseñarlos con cierto enfoque o estrategia.

Si bien podemos pensar que los objetivos del *Testing* Automatizado son triviales, pueden también diferir mucho de una organización a otra. Por mencionar algunos objetivos interesantes como para comentar:

1. Asegurar un *testing* consistente y repetible, que se ejecuten siempre exactamente las mismas acciones, con los mismos datos y verificando cada una de las cosas que siempre hay que verificar, tanto a nivel de interfaz como a nivel de base de datos.
2. Correr casos de prueba desatendidos.
3. Encontrar errores de regresión, a menor costo y en forma más temprana.
4. Correr casos de prueba más seguido (después de cada *commit* al repositorio de código por ejemplo).
5. Mejorar la calidad del *software* (más *testing*, más oportunidades de mejora) y así incrementar la confianza de los usuarios.
6. Medir *performance*.
7. Testear en diferentes sistemas operativos, navegadores, configuraciones, gestores de bases de datos, etc., sin duplicar el costo de ejecución.
8. Disminuir el tiempo de salida al mercado/correr las pruebas más rápido.
9. Mejorar la moral en los *testers*, haciendo que las tareas rutinarias sean ejecutadas en forma automática.
10. Reducir la deuda técnica. Esto va a implicar cambios en el código y vamos a querer mantener la funcionalidad intacta. Entonces, si vamos a trabajar en

mejorar la calidad interna sin afectar la calidad externa, las pruebas automáticas nos pueden ayudar a trabajar con más confianza.

Creemos que, si bien hay varias parecidas y algunas no aplican específicamente a automatización, es preciso antes de comenzar con cualquier proyecto de automatización preguntarse cuál de estos objetivos queremos lograr y cómo medir si nos acercamos a los objetivos o no.

Algunos otros ejemplos interesantes para agregar a esa lista:

11. Seguir un enfoque de integración continua y, de esa forma, tener una detección temprana de defectos. Entonces, vamos a tener un conjunto de casos de prueba que ejecutaremos todas las noches.
12. Contar con un conjunto de casos de prueba a ejecutar ante cada nueva versión del producto.
13. Tener pruebas básicas, que nos sirvan a modo de pruebas de humo, y de esa forma saber si la versión liberada a *testing* es válida para probar o se “prende fuego” muy fácil.
14. Asegurar que los incidentes reportados no vuelvan al cliente. Si se reporta un error, este se corrige y se automatiza una prueba que verifique que ese error no está presente. ¿A quién no le ha pasado que un cliente reporte un error, se corrija y a los dos meses el cliente se queje furioso de que vuelve a pasar aquel error que ya había reportado?

Ninguna de estas posibilidades es excluyente, son incluso complementarias.

También se plantea que la automatización permite encontrar errores que no serían fáciles de encontrar manualmente. Por ejemplo, errores de memoria que se dan luego de ejecutar muchas veces una funcionalidad, y ni que hablar si nos referimos a pruebas de concurrencia o *performance*.

Hay muchas cosas que solo al probar “manualmente” se pueden ver y otras que tal vez es más probable que las vea una prueba automatizada. Por ejemplo, si queremos verificar que cada elemento pedido al servidor retorne un código de respuesta sin error (en el caso de *http* sería que no devuelva ningún *404* o *500*, por ejemplo) o si queremos ver que todas las URL estén configuradas con *https*, se

puede programar y automatizar, para que en todas las pruebas se verifique y, en cambio, una persona tal vez no le preste atención en cada ocasión.

Y así como es importante definir los objetivos, es igual de importante conservarlos en mente. El peligro está, por ejemplo, en que el encargado de automatizar sea una persona que sepa programar (generalmente) y, por lo tanto, será muy probable que al usar la herramienta la encuentra desafiante y entretenida técnicamente y acabe automatizando muchas funcionalidades sin analizar de antemano qué tan relevantes son para avanzar hacia los objetivos.

Priorización: decidir qué y cuándo automatizar

Luego, con el objetivo en mente, será más fácil determinar cuáles son los casos de prueba que quiero seleccionar para automatizar. Para esto, nos basamos en algo llamado *Testing Basado en Riesgos*. En esta estrategia de pruebas se le da más prioridad a probar aquellas cosas que tienen mayor riesgo, pues si luego fallan son las que más costos y consecuencias negativas nos darán.

Con esta consideración, resulta importante hacer un análisis de riesgo para decidir qué casos de prueba automatizar, considerando distintos factores como, por ejemplo:

- Importancia o criticidad para la operativa del negocio.
- Costo del impacto de errores.
- Probabilidad de que haya fallos (esto hay que preguntárselo a los desarrolladores, que seguramente sabrán que determinado módulo lo tuvieron que entregar en menor tiempo del que necesitaban, entonces ellos mismos dudan de la estabilidad o calidad de este desarrollo).
- Acuerdos de nivel de servicio, o SLA de su sigla en inglés: *Service Level Agreement*.
- Dinero o vidas en juego, que por más que suena dramático, sabemos que hay muchos sistemas que manejan muchas cosas que implican gran sensibilidad.

Al pensar en la probabilidad de que aparezcan fallos, también hay que pensar en la forma de uso. No solo pensar en cuáles son las funcionalidades más usadas, sino

también cuáles son los flujos, opciones, datos, etc., más usados por los usuarios. Pero también combinándolos con la criticidad de la operación, pues una liquidación de sueldo por ejemplo se ejecuta una vez por mes, pero es muy alto el costo de un error en esa funcionalidad.

Al pensar en la criticidad de un caso de prueba, se debe hacer considerando la criticidad para el producto como un todo y no solo pensando en la siguiente liberación, pues seguramente el criterio sea distinto. Por ejemplo, para una liberación lo crítico es la funcionalidad X, pero si pensamos el producto como un todo, esa funcionalidad X quizá afecte a otras Y e Z que son críticas para el producto.

Una vez establecida la priorización de las pruebas, sería deseable revisarla cada cierto tiempo, pues, como los requisitos cambian por cambios en el negocio o en el entendimiento con el cliente, será importante reflejar esos cambios sobre la selección de los casos de prueba a ejecutar.

Por otra parte, ¿qué tanto automatizar? En cada caso será diferente, pero hay quienes recomiendan intentar comenzar con un objetivo del 10 o 15% de las pruebas de regresión y a medida que se va avanzando se podría llegar hasta un 60% aproximadamente. Para nosotros, será importante no proponerse como objetivo el automatizar el 100% de las pruebas, pues eso va en contra de la moral de cualquier *tester*.

Ahora, la selección de los casos de prueba no es lo único importante, sino el tener definido qué pasos, qué datos, qué respuesta esperada. Por lo tanto, a continuación comenzaremos a ver más en detalle este tipo de cosas.

¿Cómo automatizar?

Consideremos que ya tenemos los casos de prueba diseñados o, si no es así, utilicemos las técnicas de diseño de pruebas que presentamos en los capítulos previos de este libro.

Para comenzar desde mayor nivel de granularidad e ir refinando luego, empezaremos revisando el inventario de funcionalidades, asignando prioridades a

cada una. Luego, asignaremos prioridades a cada caso de prueba que tenemos preparado para cada una de las distintas funcionalidades. Esta organización y priorización nos servirá para dividir el trabajo (en caso de que seamos un equipo de varios *testers*) y para planificarlo, ya que, como veremos luego, se recomienda organizar los artefactos de prueba por algún criterio, como por ejemplo, agruparlos por funcionalidad.

El diseño de casos de prueba, para pruebas automatizadas, conviene que esté definido en dos niveles de abstracción. Por un lado, vamos a tener los que llamamos **casos de prueba abstractos** o **conceptuales** y por otro lado los llamados **casos de prueba con datos**. Si bien ya explicamos estos conceptos en el capítulo introductorio, hagamos un repaso y adaptación a este contexto. Los casos de prueba abstractos son guiones de prueba, que al indicar que se utilizarán datos no hacen referencia a valores concretos, sino a clases de equivalencia o a grupos de valores válidos, como podría ser *número entero entre 0 y 18*, *string de largo 5* o *identificador de cliente válido*. Por otra parte, tendremos los casos de prueba con datos, en donde se instancian los casos de prueba abstractos con valores concretos, como por ejemplo utilizar el número 17, el *string abcde* y 1.234.567-8 respectivamente. Estos últimos son los que realmente podemos ejecutar, ya que están completos. Los casos de prueba abstractos no son ejecutables hasta que se decide exactamente qué dato usar, cosa que lo convierte en un caso de prueba concreto.

Nos interesa hacer esta distinción entre esos dos “niveles”, pues en distintos momentos de la automatización vamos a estar trabajando con ellos, para poder seguir un enfoque de *Data-Driven Testing*, que realmente plantea una gran diferencia con respecto al *simple scripting*.

Para *scripts* de *Testing Automatizado*, *Data-Driven Testing*, implica probar la aplicación utilizando datos tomados de una fuente de datos, como una tabla, archivo, base de datos, etc., en lugar de tener los mismos fijos en el *script*. En otras palabras, parametrizar el *test case*, permitir que se pueda ejecutar con distintos datos. El objetivo principal es poder incorporar más casos de prueba simplemente agregando más datos.

Entonces, antes que nada vamos a grabar el caso de prueba con la herramienta. Como, para grabar es preciso ejecutar sobre la aplicación, entonces, vamos a necesitar un caso de prueba con datos. El *script* resultante corresponderá a un caso de prueba con datos y al ejecutarlo usará exactamente el mismo caso de prueba, con los mismos datos. El siguiente paso será parametrizar esa prueba, de modo que el *script* no use los datos concretos grabados sino que los tome de una fuente de datos externa (puede ser de un archivo, una tabla, etc.). En ese momento, el *script* se corresponde con un caso de prueba abstracto y en la fuente de datos externa (llamémosle *datapool*) cargaremos distintos valores interesantes del mismo grupo de datos. De esta forma, nuestro *script* automatizado nos permite ejecutar el mismo caso de prueba abstracto con diferentes datos concretos, sin necesidad de grabar cada caso de prueba con datos.

Por otra parte, debemos pensar en el **oráculo** de la prueba. Cuando se define un caso de prueba el *tester* expresa las acciones y los datos a utilizar en la ejecución, pero ¿qué pasa con el oráculo? ¿Cómo determinamos que el resultado del caso de prueba es válido o no? Es necesario definir (diseñar) las acciones de validación que nos permitan dejar plasmado un oráculo capaz de determinar si el comportamiento del sistema es correcto o no. Hay que agregar las validaciones suficientes para dar el veredicto en cada paso, persiguiendo también el objetivo de evitar tanto los falsos positivos como los falsos negativos (estos conceptos y cómo evitar los problemas asociados que traen, los veremos en la siguiente sección).

Diseño de *suites* de prueba

Generalmente todas las herramientas nos permiten agrupar los casos de prueba, de manera de tenerlos organizados, y ejecutarlos en conjunto. La organización la podemos definir por distintos criterios, dentro de los cuales podemos considerar:

- Módulo o funcionalidad: agrupando todos los casos de prueba que actúan sobre una misma funcionalidad.
- Criticidad: podríamos definir un grupo de pruebas que se debe ejecutar siempre (en cada *build*), ya que son las más críticas, otro de nivel medio, que lo ejecutamos con menos frecuencia (o tal vez que se seleccionan solo si hay

cambios en determinadas funcionalidades) y uno de poca criticidad, que ejecutaríamos opcionalmente si contamos con tiempo para hacerlo (o cuando cerremos un ciclo de desarrollo y queramos ejecutar todas las pruebas disponibles).

Estos criterios incluso se podrían combinar, teniendo criterios cruzados o anidados. En muchas herramientas en lugar de solo definir *suites* también podremos definir *tags* (etiquetas) y de esa forma tener distintas agrupaciones por diferentes criterios, lo cual será útil para poder ejecutar subconjuntos de nuestras pruebas según la necesidad de cada momento. También podremos agrupar los resultados en base a estos criterios y facilitar nuestro análisis.

Por otra parte, es interesante definir dependencias entre *suites*, ya que existen determinadas funcionalidades que, si fallan, directamente invalidan otros test. No tiene sentido “perder tiempo” ejecutando pruebas que sabemos que van a fallar. O sea, ¿para qué ejecutarlas si no van a aportar información? Es preferible frenar todo cuando se encuentra un problema, atacar el problema y volver a ejecutar hasta lograr que todo quede funcionando (esto va de acuerdo con la metodología *Jidoka*³⁹, por si quieren investigar más al respecto).

³⁹ Información sobre Jidoka: <<http://en.wikipedia.org/wiki/Autonomation>>

Buenas prácticas de automatización de pruebas

Con lo visto hasta ahora, ya somos capaces de guiar nuestros primeros pasos de automatización de pruebas, pero es cierto que cuando comencemos a profundizar en el asunto, nos iremos enfrentando a un montón de situaciones más complejas o de resolución no tan evidente. En esta sección intentaremos mostrar algunos enfoques para resolver muchas situaciones que hemos tenido que enfrentar alguna vez, basándonos en nuestra literatura favorita y en la experiencia de nuestro trabajo.

Las cosas claras al comenzar

A lo largo de la vida de un producto vamos a tener que mantener el set de pruebas que vayamos automatizando. Cuando tenemos un conjunto de pruebas reducido, no es difícil encontrar la prueba que queramos ajustar cuando haga falta, pero cuando el conjunto de pruebas comienza a crecer, todo se puede volver un lío. Es muy importante, entonces, definir y dejar clara la organización y nomenclatura a utilizar, que a futuro nos permita manejar lo más simple posible el enorme set de pruebas que vamos a tener.

Nomenclatura

Es importante definir una nomenclatura de casos de prueba y carpetas (o lo que brinde la herramienta de automatización para organizar las pruebas). Esta práctica, si bien es simple, redundante en muchos beneficios. Algunas recomendaciones del estilo:

- Utilizar nombres para los casos de prueba, de forma tal que sea fácil distinguir aquellos que ejecutaremos de aquellos que son partes de los casos de prueba principales (si se sigue una estrategia de modularización, ¡también recomendada!). A los casos de prueba que efectivamente

ejecutaremos, que podríamos incluso considerar los ciclos funcionales que invocan a distintos casos de prueba más atómicos, se los puede nombrar con *Ciclo XX*, donde *XX* generalmente será el nombre de la entidad o acción más importante relacionada a la prueba.

- Por otro lado, es útil definir una estructura de carpetas que permita separar los casos de prueba generales (típicamente *login*, acceso a los menús, etc.) de los casos de prueba de los distintos módulos. Esto es para promover la reutilización de casos de prueba que se diseñaron, de forma tal que es fácil incluirlos en otros casos.
- Muchas veces también surge la necesidad de crear casos de prueba temporales, que luego serán eliminados, a los cuales se los puede nombrar con un prefijo común como *pru* o *tmp*.

Esto va más asociado a los gustos o necesidades de cada uno, nuestra sugerencia es dejar esto establecido antes de comenzar a preparar *scripts* y que el repositorio comience a crecer en forma desorganizada.

Comentarios y descripciones

Cada caso de prueba y *datapool* puede tener una descripción que diga en líneas generales cuál es su objetivo. Por otro lado, dentro de los casos de prueba podemos incluir comentarios que ilustren los distintos pasos. Dentro de los *datapool* se podría agregar también una columna extra para registrar comentarios, en la cual se indique el objetivo de cada dato en concreto que se utilizará en las pruebas y se cuente qué es lo que se está intentando probar.

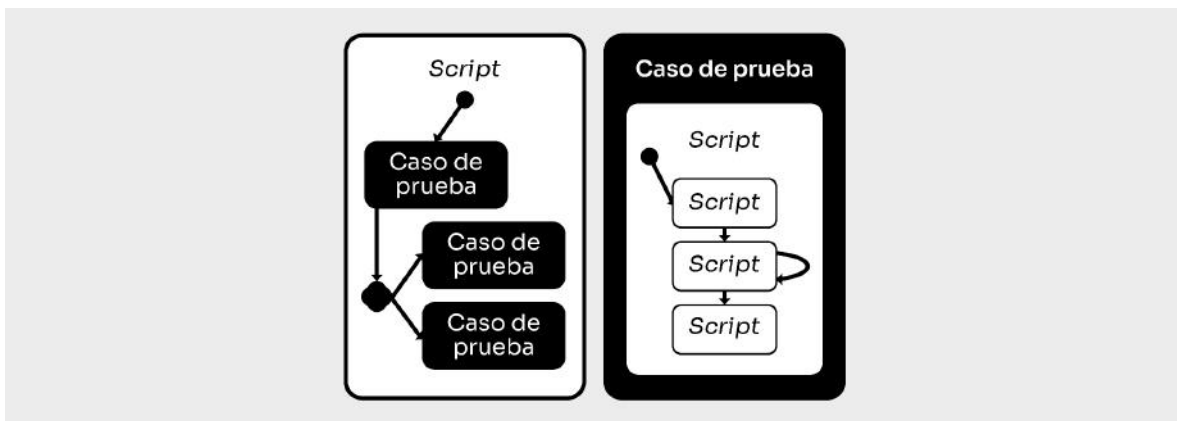
Relación entre caso de prueba y *script* automatizado

¿Cómo deberían hacerse los *scripts* en la herramienta? ¿Uno por caso de prueba?
¿Puedo hacer un *script* que pruebe distintos casos de prueba al mismo tiempo?

La siguiente figura representa las dos opciones. A la izquierda tenemos un *script* que ejecuta distintos casos de prueba. Quizá en su ejecución analiza distintas opciones sobre los datos o sobre el estado del sistema y, de acuerdo a esa evaluación, decide ejecutar un caso de prueba u otro. A la derecha, tenemos un caso

de prueba que está modularizado en distintos *scripts*. Vemos que hay distintos casos de prueba pequeños que son ejecutados por un *script* que los incluye y orquesta.

Como todo en la ingeniería de *software*, esto depende, en este caso, del caso de prueba. Algunos plantean pensar en cuántas bifurcaciones lógicas se dan en el caso de prueba. Lo más adecuado, desde nuestro punto de vista, es tomar un enfoque modular. O sea, tener distintos módulos (*scripts*) que hagan distintas partes de la prueba y luego un *script* que orqueste todas estas partes (como la opción de la derecha de la figura). De esa forma, podemos reutilizar las pequeñas partes y hacer distintas pruebas que las compongan de distintas formas.



Relación entre *scripts* y casos de prueba

En ese caso la relación sería *un caso de prueba hecho con varios scripts*.

Ventajas

- Mantenimiento: más fácil, se pueden reutilizar los módulos, se puede cambiar el flujo en distintos niveles, el *script* del caso de prueba queda más legible, pues se puede ver todo el flujo que recorre a “gran escala” y luego profundizar en las partes que interese.
- Cambiar el flujo del caso de prueba: se puede manejar más fácil el flujo del caso de prueba y hacer, por ejemplo, que cierto módulo se repita una cantidad dada de veces (fija o variable). El ejemplo típico de una factura, si modularizamos la parte donde ingresamos una línea de la factura, con cierto producto y cantidad, podemos hacer que esa parte se ejecute distinta

cantidad de veces, con el fin de probar la factura con distintas cantidades de líneas, tal como está representado con el bucle en la parte derecha de la figura.

- **Análisis de resultados:** es más sencillo analizar los reportes de resultados siempre y cuando cuidemos la nomenclatura, pensando cómo se mostrarán finalmente las distintas partes en el reporte.

Si se tiene documentación de casos de prueba (si se solían ejecutar manualmente, por ejemplo) una buena práctica podría ser llevar una matriz que relacione los casos de prueba con los distintos *scripts* involucrados. Esto permite conocer también qué verificar cuando cambien requerimientos que tengan impacto sobre ciertas pruebas y así, consecuentemente, sobre algunos *scripts*.

Una alternativa que se plantea generalmente es la de diseñar los casos de prueba de manera secuencial en caso de que los resultados sean deterministas. Si los resultados tienen alguna variabilidad no definida de antemano, entonces, se pueden agregar distintos flujos, pero lo mejor es **mantener las cosas simples y secuenciales**. Muchas veces, los que venimos “del palo” de la programación tendemos a hacer casos de prueba tan genéricos (que cubran todos los casos) que terminan siendo muy complejos.

Si diseñamos un único caso de prueba que contempla distintas opciones, probablemente sea más difícil de entender y, para eso, tengamos que analizar qué se está consultando en cada decisión (bifurcación), qué se hace en un flujo u otro, etc., a menos que seamos muy cuidadosos y llenemos el *test case* de comentarios que simplifiquen ese análisis. De todos modos, un *test case* secuencial con un nombre descriptivo da la pauta de qué es y qué hace.

Por otra parte, si el día de mañana agregamos un nuevo caso, ¿dónde lo hago? ¿Cómo agrego la bifurcación? ¿Cómo manejo los datos asociados? Si en cambio hacemos un *test case* nuevo, secuencial, con un *datapool* para ese caso, simplifica bastante esa tarea.

¿Cómo evitar falsos positivos y falsos negativos?

Cuando hablamos de automatización, uno de los puntos más delicados es el resultado “mentiroso”, el también conocido como *falso positivo* y *falso negativo*. Quienes ya han automatizado saben que esto es un problema y a quienes van a empezar les adelantamos que van a tener este tipo de problemas. ¿Qué podemos hacer para evitarlo? ¿Qué podemos hacer para contribuir a que el caso de prueba haga lo que se supone que debe hacer? ¿Eso no les suena a *testing*?

Estas definiciones provienen del área de la medicina:

- *Falso positivo*: un estudio indica enfermedad cuando no la hay.
- *Falso negativo*: un estudio indica que todo está normal cuando el paciente está enfermo.

Llevándolo a nuestro campo, podríamos decir que:

- ***Falso positivo***: la ejecución de una prueba donde a pesar de que el sistema bajo pruebas ejecuta correctamente, el test nos dice que hay un error (que hay una enfermedad). Esto agrega mucho costo, pues el *tester* ante el resultado buscará (sin sentido) dónde está el *bug*.
- ***Falso negativo***: la ejecución de una prueba que no muestra un fallo a pesar de que la aplicación tiene un *bug*. Tanto esto como el falso positivo pueden darse, por ejemplo, por un estado inicial incorrecto de la base de datos o problemas en la configuración del entorno de pruebas. Otra causa común es la de no tener las validaciones adecuadas (un caso que ejecuta, quizá el error se hace visible, pero como no había una validación, entonces, no se da un veredicto correcto).

Si creemos que el falso positivo nos genera problemas por agregar costos, con un falso negativo ¡hay errores y nosotros tan tranquilos!, confiados en que esas funcionalidades están cubiertas, que están siendo probadas y, por lo tanto, que no tienen errores.

Obviamente, ¡queremos evitar que los resultados nos “mientan”! No nos caen bien los mentirosos. Lo que se espera de un caso de prueba automático es que el resultado sea fiel y no estar perdiendo el tiempo en verificar si el resultado es correcto o no.

No queda otra que hacer un análisis proactivo, verificando la calidad de nuestras pruebas y anticipando a situaciones de errores. O sea, “ponerle un pienso” a los casos de prueba y que no sea solo *record and playback*.

Algo muy importante es prestar atención a lo que se conoce como *ambientes de prueba*. En el contexto de desarrollo de *software*, un *ambiente* se refiere a un entorno que proporciona las condiciones necesarias para realizar actividades específicas, como desarrollo o pruebas, incluso tendremos *el ambiente de producción*. Cada ambiente estará configurado de manera única y estará aislado del resto para cumplir con sus objetivos. Por ejemplo, el ambiente de producción será óptimo y potente para poder dar un buen servicio a los usuarios, el de pruebas podrá ser de menor potencia, pero replicando configuraciones del entorno de producción para propiciar que se identifiquen problemas que puedan afectar a los usuarios, y el de desarrollo estará aislado y será ágil y acotado para facilitar la experimentación iterativa que se busca en el desarrollo.

Entonces, para bajar el riesgo de que ocurran problemas en las pruebas automáticas, deberíamos tener un ambiente controlado, que esté solo disponible para la ejecución de estas pruebas. Entonces, ya nos vamos a ahorrar dolores de cabeza y el no poder reproducir problemas detectados por las pruebas, ya que, si los datos o configuraciones de ambiente cambian constantemente, no sabremos qué es lo que pasa.

Por otra parte, ¿deberíamos probar los propios casos de prueba! ¿Quién nos asegura que estén bien programados? Al fin de cuentas, el código de pruebas es código también, y como tal, puede tener fallas y oportunidades de mejora. Y quién mejor que nosotros, *testers*, para probarlos.

Probando en busca de falsos positivos

Si el *software* está sano (siguiendo la analogía con la medicina) y no queremos que se muestren errores, debemos asegurarnos de que la prueba esté probando lo que

quiere probar y esto implica verificar las condiciones iniciales tanto como las finales. O sea, un caso de prueba intenta ejecutar determinado conjunto de acciones con ciertos datos de entrada para verificar los datos de salida y el estado final, pero es muy importante (y especialmente cuando el sistema que estamos probando usa bases de datos) asegurar que el estado inicial es el que esperamos.

Entonces, si, por ejemplo, vamos a crear una instancia de determinada entidad en el sistema, la prueba debería verificar si ese dato ya existe antes de comenzar la ejecución de las acciones a probar, pues, si es así, la prueba fallará (por clave duplicada o similar) y en realidad el problema no es del sistema, sino de los datos de prueba. Dos opciones: verificamos si existe y, si es así, ya utilizamos ese dato, y si no, damos la prueba como concluida dando un resultado de *inconcluso* (¿o acaso los únicos resultados posibles de un test son *pass* y *fail*?).

Si nos aseguramos de que las distintas cosas que pueden afectar el resultado están en su sitio, tal como las esperamos, entonces vamos a reducir mucho el porcentaje de errores que no son errores.

Probando en busca de falsos negativos

Si el *software* está enfermo, la prueba debe fallar! Una posible forma de detectar los falsos negativos es insertar errores al *software* y verificar que el caso de prueba encuentre la falla. Esto, en cierta forma, sigue la idea del *Testing* de Mutación. Es muy difícil cuando no se trabaja con el desarrollador directamente como para que nos ingrese los errores en el sistema, ya que es muy costoso preparar cada error, compilarlo, hacer *deploy*, etc. y verificar que el test encuentra ese error. Muchas veces se puede hacer variando los datos de prueba o jugando con distintas cosas. Por ejemplo: si tenemos de entrada un archivo de texto plano, cambiamos algo en el contenido del archivo para obligar a que la prueba falle y verificamos que el caso de prueba automático encuentre esa falla. En una aplicación parametrizable, también se puede lograr modificando algún parámetro.

Siempre la idea es verificar que el caso de prueba se puede dar cuenta del error y, por eso, intentamos con estas modificaciones hacer que falle. De todos modos, lo que al menos podríamos hacer es pensarlo: ¿qué pasa si el *software* falla en este

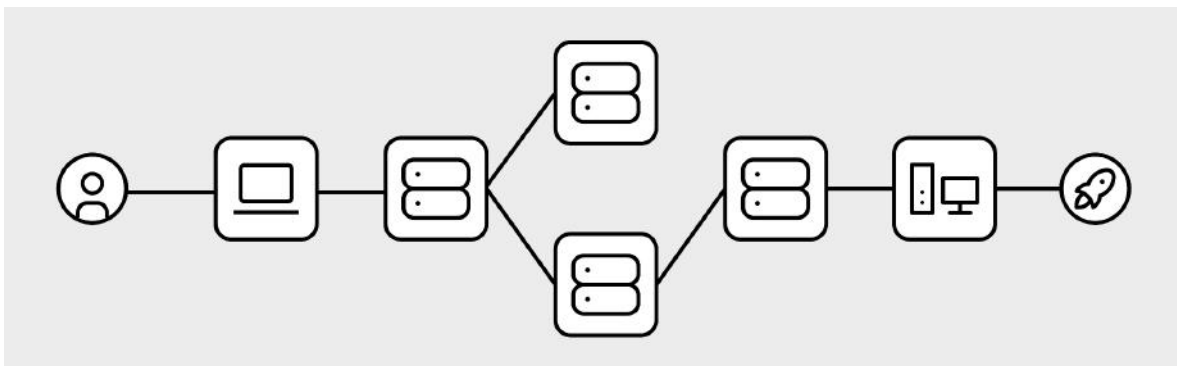
punto?, ¿este caso de prueba se daría cuenta o nos hace falta agregar alguna otra validación?

Ambas estrategias nos van a permitir tener casos de pruebas más robustos, pero, ¡ojo!, ¿quizá luego sean más difíciles de mantener? Por supuesto que no vamos a hacer esto con todos los casos de prueba que automaticemos, será aplicable a los más críticos, a los que realmente nos merezca la pena o quizá a los que sabemos que a cada poco nos dan problemas.

Pruebas de sistemas que interactúan con sistemas externos

¿Qué pasa si mi aplicación se comunica con otras aplicaciones mediante mecanismos complejos? ¿Qué pasa si consume servicios *web* expuestos en otros servidores? ¿Qué pasa si mi aplicación tiene lógica muy compleja? ¿Puedo automatizar mis pruebas en estas situaciones?

Para dar respuesta a estas preguntas, veamos lo que se representa en la siguiente figura. A partir de un botón en la aplicación bajo pruebas, se ejecuta una lógica compleja, hay comunicación con varias aplicaciones externas y se dispara un cohete.



Pruebas de sistemas con interacción con sistemas externos

Las herramientas de automatización (al menos en las que nos estamos enfocando aquí) tienen como objetivo reproducir la interacción del usuario con el sistema, por

lo tanto, estas complejidades de fondo son *casi* indiferentes. Una vez que el usuario presiona un botón, la lógica que se ejecuta a partir de esa acción puede ser simple o compleja, pero *a la vista* de la herramienta eso está oculto (tan oculto como lo está a la vista del usuario). Tal como se muestra en la figura, no importa si se dispara un cohete o lo que fuera, lo que *importa* para automatizar es la interfaz de usuario en este caso.

Ahora bien, hay veces en que el caso de prueba requiere que se hagan otras acciones que no se pueden hacer en el navegador o en la interfaz gráfica del sistema bajo pruebas, como por ejemplo consultar una base de datos, copiar archivos a un determinado lugar, etc. Para estas acciones, las herramientas generalmente brindan la posibilidad de realizarlas ejecutando alguna funcionalidad especial o programando en algún lenguaje de propósito general.

El hecho de que una aplicación con una lógica compleja no agregue dificultades a la automatización no significa que no agregue dificultades al momento de pensar y diseñar las pruebas. Dos de los aspectos en que más se pueden complicar son la preparación de los datos y la simulación de los servicios externos que se consumen. En particular, en el segundo punto hay veces en que es interesante que realmente el sistema bajo pruebas se *conecte* al servicio externo y hay otras veces que es preferible simular este servicio, incluso probar la interacción con este. Al artefacto que simula el servicio externo generalmente se lo llama *Stub* o *Mock Service* y existen herramientas que permiten implementarlos fácilmente. Por ejemplo, en el caso de que ese servicio sea un *web service*, podrían considerar la herramienta SoapUI⁴⁰, que brinda una interfaz muy amigable para generar *Mock Services*, así como para testear servicios *web*.

⁴⁰ SoapUI: <<http://soapui.org>>

Considerar la automatización en la planificación del proyecto

Mucha gente sigue con la concepción de que el *testing* es algo que hay que dejar para el final, si es que da el tiempo, y en realidad es una tarea que hay que hacerla y planificarla desde el comienzo.

En cuanto a la automatización, algunas de las tareas a planificar son las que se ven a continuación:

- Automatización
- Mantenimiento
- Ejecuciones
- Verificación y reporte de *bugs*
- Correcciones de *bugs* detectados

Hay que ver cuándo comenzar a automatizar (si desde el comienzo o luego de cierta etapa en la que se espera lograr una versión estable de la aplicación) y pensar en el mantenimiento que nos va a llevar todo eso también.

Buenas prácticas de diseño de casos automatizados

En una edición de la revista *Testing Experience* (que lamentablemente ya no está disponible) hay un interesante artículo sobre principios para tener en cuenta al momento de automatizar test funcionales. Queremos compartir aquí también los apuntes principales para tenerlos bien presentes.

1. Primero, diseñar las pruebas. Evitar la creación de pruebas *on the fly*. Nuestra *suite* de test debe crecer de forma ordenada y no a medida se nos van ocurriendo casos sin ton ni son.

2. No automatizar todo. Ciertos test pueden ser muy costosos de automatizar y quizá fácilmente ejecutados en forma manual, para evitar el alto costo de automatización, pensando, más que nada, en su mantenimiento.

3. Escribir pruebas cortas. En situaciones en las que una o más test fallen, cualquier miembro del equipo debe ser capaz de *trackear* la causa del error. Un mecanismo útil para esto es usar BDD (*Behavior-Driven Development*), lo que veremos en la siguiente sección.

4. Crear pruebas independientes. Evitar el acoplamiento y aumentar la cohesión de los test. No puede pasar que, si ejecutamos los test en diferente orden, entonces, los resultados sean diferentes, o si ejecutamos el mismo test más de una vez, entonces, falle. Debemos pensar mucho en cómo diseñar los datos de prueba y los distintos pasos, de forma modular y autosuficiente.

5. Enfocarse sobre *readability*. El código fuente de cada test debería ser *self-explanatory*. Si el código es fácil de leer, vamos a necesitar menos esfuerzo en documentar o en agregar comentarios extensos para que pueda ser entendido por otras personas.

6. Los test deben ser rápidos. El *Testing* Funcional Automatizado debe ser un indicador rápido de la calidad de la aplicación.

7. Crear pruebas resistentes al cambio. Quizás esta sea una de las principales desventajas de las pruebas funcionales automatizadas.

8. Los test automatizados no pueden reemplazar a los humanos. Este no debe ser el único *testing* que se ejecuta. Otras técnicas, con la intervención de los *testers* humanos, deben ser aplicadas sobre el *software* en favor de generar más conocimiento.

BDD (*Behavior-Driven Development*)

BDD refiere a *Behavior Driven Development*, o sea, desarrollo dirigido por comportamiento. Como bien lo indica su nombre, no se trata de una técnica de *testing*, sino que es una estrategia de desarrollo (así como TDD, que es *Test Driven Development*). Lo que plantea es definir un lenguaje común para el negocio y para los técnicos y utilizar eso como parte inicial del desarrollo y el *testing*. Por esto, es importante que cada *tester* entienda bien qué es BDD. Por ejemplo, podríamos definir un requerimiento y los criterios de aceptación en un lenguaje que, por un lado, deje claro a desarrollo qué es lo que hay que programar y, por otro lado, que sirva de insumo para *testing* para entender qué probar o, incluso, que sirva como base para el test automatizado. Veremos más de esto a continuación.

BDD y TDD

Nos gusta ver BDD como una evolución del TDD, más a alto nivel. En TDD se enfoca a la prueba unitaria, en cambio en BDD se enfoca en la prueba de más alto nivel, la prueba funcional, la de aceptación, el foco está en cumplir con el negocio y no solo con el código.

Ambos enfoques podrían convivir, ya que se podría especificar una prueba de aceptación y luego refinar en pruebas unitarias al momento de codificar esa funcionalidad en las distintas capas. Tal vez, una ventaja clara de BDD es que desarrollo se enfoca en ver qué es lo que tiene que funcionar y de qué forma a nivel de negocio.

BDD en metodologías ágiles

Esta estrategia aplica bien en las metodologías ágiles, ya que generalmente en ellas se especifican los requerimientos como historias de usuario. Estas historias de usuario deberán tener sus criterios de aceptación y de ahí se derivan las pruebas de aceptación, las cuales pueden ser escritas directamente en lenguaje Gherkin, que lo explicamos a continuación.

Lenguaje común para negocio y técnicos: Gherkin

Gherkin es un lenguaje común, que puede escribir alguien sin conocimientos en programación, también lo puede comprender un programa, de forma tal que se puede utilizar como especificación de pruebas.

Típicamente, estas pruebas se van a guardar en archivos “.feature”, los cuales deberían estar versionados junto al código fuente del sistema que se está probando. Este es un ejemplo simple tomado de Cucumber⁴¹:

```
1: Feature: Some terse yet descriptive text of what is desired
2:   Textual description of the business value of this feature
3:   Business rules that govern the scope of the feature
4:   Any additional information that will make the feature easier to understand
5:
6: Scenario: Some determinable business situation
7:   Given some precondition
8:     And some other precondition
9:   When some action by the actor
10:    And some other action
11:    And yet another action
12:   Then some testable outcome is achieved
13:    And something else we can check happens too
14:
15: Scenario: A different situation
16:   ...
```

⁴¹ Cucumber: <<https://cucumber.io>>

En estos archivos se especifica:

- **Feature:** nombre de la funcionalidad que vamos a probar, el título de la prueba.
- **Scenario:** habrá uno por cada prueba que se quiera especificar para esta funcionalidad.
- **Given:** acá se marca el contexto, las precondiciones.
- **When:** se especifican las acciones que se van a ejecutar.
- **Then:** acá se especifica el resultado esperado, las validaciones a realizar.

Puede haber un *feature* por archivo y este contendrá distintos escenarios de prueba.

Gherkin soporta muchos lenguajes, así que, si bien lo más común es verlo en inglés, se podría hacer en español también.

¿Quién escribe las pruebas de aceptación?

Esto depende mucho de cada equipo, pero podrá ser alguna persona encargada del análisis de requisitos, del *testing* o incluso del desarrollo. Tal vez una buena opción es hacerlo en conjunto, al menos el *brainstorming*, como parte del refinamiento de las historias. Luego, el *tester* se encargaría de asegurarse de que estén bien escritas, completas, con el cubrimiento de acuerdo a la estrategia de pruebas definidas.

BDD para automatizar pruebas

¿Por qué estamos hablando de esta técnica de desarrollo en este capítulo que habla de automatización de pruebas? El lenguaje Gherkin es simplemente texto con algunas palabras clave y algo de estructura. Hay herramientas, como Cucumber, que permiten implementar una capa de conexión entre esa especificación de prueba y la interfaz del sistema que se quiere probar; así, es posible utilizar eso como los pasos de una prueba automatizada.

Una buena receta para trabajar BDD podría ser:

- Escribir las pruebas de aceptación en Gherkin (puede hacerlo una persona sin conocimiento en programación), cuando se refinan las historias de usuario.
- Trabajar durante el *sprint* en el desarrollo. Al inicio del *sprint*, mientras no hay ninguna historia pronta para probar, el *tester* que automatiza puede trabajar en quitar deuda técnica⁴² de *testing*.
- Realizar test exploratorio una vez que una historia esté lista para probar y, si se considera que se puede automatizar, se implementa la capa que conecta el archivo *feature* con la interfaz del sistema a probar (para cada línea del archivo *feature* con los criterios de aceptación, se especifica cómo se ejecuta sobre el sistema). Esto es usando Cucumber con algún lenguaje (Java, Ruby, etc.), con algún *driver* apropiado, que puede ser Selenium, si probáramos una *web*; Appium, si es algo mobile; SoapUI, para servicios *web*.
- Versionar el código de las pruebas, el código del sistema y las *features* con el mismo criterio.

⁴² *La deuda técnica* es una analogía en el desarrollo de *software* que describe el costo acumulado de elegir soluciones rápidas y simples en el presente, que pueden resultar en problemas o complicaciones en el futuro. Similar a *la deuda financiera*, la deuda técnica requiere ser “pagada” eventualmente mediante la corrección y mejora del código o cambios de decisiones de diseño, con el fin de mantener la integridad y eficiencia del sistema a largo plazo.

Ejecución de pruebas automáticas

Utilizar herramientas *Record and Playback* suena como algo fácil, pero como ya hemos visto, tenemos que tener en cuenta varias consideraciones para el momento previo al *playback*. Ahora veremos que también hay algunos aspectos importantes a tener en cuenta para el momento del *playback*.

Ambientes de prueba

Es fundamental hacer una buena gestión de los ambientes de prueba. Dentro de esa gestión, tenemos que considerar muchos elementos que son parte del ambiente:

- Los fuentes y ejecutables de la aplicación bajo pruebas.
- Los artefactos de prueba y los datos que estos usan.
- A su vez, los datos están relacionados con la base de datos del sistema bajo pruebas, con lo que tendremos que gestionar el esquema y los datos de la base de datos que se corresponden con el entorno de pruebas.
- Las conexiones con las dependencias externas (si el sistema usa canales de pago, ¿cómo se va a gestionar eso en el ambiente de prueba? Generalmente se configura para usar los ambientes de prueba provistos por los proveedores de esas dependencias externas).

Agreguemos la complicación de que tal vez tengamos distintas pruebas a realizar con distintas configuraciones, parametrizaciones, etc. Entonces, para esto tenemos más de un ambiente de pruebas o un ambiente y muchos respaldos (o del inglés, *backups*) de base de datos, uno para cada conjunto de pruebas. La complejidad extra que agrega esto es que hay que realizar un mantenimiento específico a cada *backup* (por ejemplo, cada vez que haya cambios en el sistema en los que se modifique la base de datos, será necesario impactar a cada *backup* con esos cambios).

Pero, si uno de estos elementos no está en sintonía con el resto, probablemente las pruebas fallen y no estaremos siendo eficientes con nuestros esfuerzos. Es importante asegurar que, cada vez que una prueba reporte un fallo, sea porque hay un *bug* y no generemos falsas alarmas.

¿Cuándo ejecutar las pruebas, quién y dónde?

Pasemos ahora a otro punto que no tiene tanto que ver con los detalles “técnicos” del *testing*. Veamos temas de planificación. Es necesario planificar las ejecuciones, pero no solo eso. Lo ideal es que el *testing* sea considerado desde el comienzo (sí, hay que decirlo muchas veces para que quede claro) y si se desea automatizar, considerar las tareas asociadas también desde el comienzo.

¿Cuándo ejecutar? Lo primero que uno piensa es *tan frecuente como se pueda*. Sin embargo, los recursos son escasos y, dependiendo de la cantidad de pruebas automatizadas, el tiempo que se demore en ejecutar puede ser muy prolongado. Por otro lado, muchas veces las ejecuciones tienen costo, ya sea de las herramientas de automatización (muchas de las herramientas comerciales cobran por cantidad de ejecuciones) o si se usan plataformas Cloud para distintas cosas como, por ejemplo, para tener ejecución en diferentes entornos de sistemas operativos, *browsers* o dispositivos.

Se podría tomar la decisión en base a este pseudo-algoritmo:

- Si tenemos pocos test, o se ejecutan en poco tiempo, entonces, ejecutarlos **todos**.
- Si ejecutan en mucho tiempo, entonces, para seleccionar qué ejecutar:
 - o considerar prioridad en base a riesgo.
 - o considerar análisis de impacto (en base a los cambios de la nueva versión a probar).

Pensemos que mayor cantidad de ejecuciones significa más retorno de la inversión (ROI) de la automatización. Además, no alcanza con testear, también hay que corregir y ese esfuerzo se debe considerar al planificar.

El conjunto de pruebas que queremos ejecutar puede variar en cada ambiente, así como la frecuencia con la que las ejecutamos. Por ejemplo (y esto puede variar según preferencias o estilos de cada equipo):

- Desarrollo querrá ejecutar las pruebas relevantes a lo que están desarrollando de manera muy frecuente, para obtener *feedback* temprano y rápido, en su ambiente local o de desarrollo.
- Luego, si se sigue un esquema de **Integración y Entrega Continua**⁴³ (**conocido como CI/CD por sus siglas en inglés**), habrá una herramienta que tome los últimos cambios en el repositorio de código, cree un ambiente nuevo y limpio para pruebas de integración, y ejecute una serie de pruebas según cómo esté configurado ese proceso, al que se le suele llamar *pipeline* (haciendo analogía con una cañería con diferentes filtros en su transcurso). Esto ayuda a capturar ese tipo de errores sobre los que, luego, desarrollo dice: “en mi máquina anda”, ya que se están ejecutando las pruebas en otras máquinas.
- Si este automatismo resulta exitoso (o sea, todos los chequeos dan un veredicto positivo), entonces, la nueva versión de la aplicación quedaría disponible para ser probada manualmente en algún ambiente de *testing*.

Una estrategia que se usa en ocasiones es ejecutar el conjunto de pruebas completo durante la noche, así de mañana, cuando llegan los desarrolladores, disponen de una lista de posibles asuntos a solucionar y un *feedback* de los cambios que introdujeron el día anterior. Mientras menos tiempo pase entre sus cambios y el resultado de las pruebas, más rápido lo solucionarán. Esto permitiría evitar que pasen a *testing* cosas que no funcionen.

Luego, cuando se pasa la aplicación a *testing*, se debería correr un conjunto más grande de pruebas de regresión que intenten asegurar que los errores que se hayan reportado y marcado como solucionados no estén presentes en esta nueva versión. Este conjunto de pruebas puede tardar más tiempo en ejecutar. Estas pruebas no deben ser necesariamente periódicas, sino que pueden adecuarse al cronograma del proyecto con las fechas previstas de liberación.

⁴³ Recomendamos leer *Continuous delivery*, Jez Humble and David Farley
<<https://www.goodreads.com/book/show/8686650-continuous-delivery>>

Cuando se llega a la versión entregable de la aplicación (considerando que se cumplen los criterios de aceptación y acuerdos establecidos), se libera para el cliente. Según el tipo de sistema, a veces el cliente primero prueba en un ambiente de preproducción, el cual debería ser totalmente simétrico en configuración al de producción (esta es la diferencia con el ambiente de *testing* del equipo de desarrollo). Aquí también aporta valor tener un conjunto de pruebas automatizadas para ejecutar antes que el cliente y podría ser usado para brindar evidencia si hace falta presentar documentación.

Este conjunto de pruebas podría ser como mínimo el que se ejecutó en *testing* e incluso se le podría entregar al cliente el conjunto de casos de prueba automatizados junto con la aplicación liberada, ya que le da mayor seguridad y confianza al cliente el saber que se testeó antes de su liberación.

¿Qué hacemos con un *bug*?

Si un reporte nos dice que hubo un fallo, lo primero es dudar de la prueba. O sea, determinar si el fallo se debió a una falla en el sistema o si en realidad hubo algún problema con el caso de prueba. Es muy recomendable asegurarse de no reportar un error que no existe. Eso mejora las relaciones.

Para esto, podemos verificar el supuesto error manualmente, para ver si realmente se trata de un error, y en ese caso ver si se puede identificar la causa, si ocurre siempre, con todos los datos o solo algunos, etc. Luego, hay que intentar encontrar la cantidad de pasos mínima para reportarlo, ya que mientras más fácil sea encontrar la causa que lo genera, va a ser más fácil de resolver para quien desarrolle y más fácil reproducir esa situación, como para darse cuenta de si lo está resolviendo o no.

Lo que sigue es reportarlo en el sistema gestor de incidentes, tal como se hace con los *bugs* encontrados con la ejecución manual de pruebas.

Estrategia de automatización

Ahora que ya conocemos mucho más sobre automatización en *testing*, hagamos un repaso de qué aspectos hay que pensar y definir para diseñar una estrategia de automatización adecuada:

- **Objetivos:** antes de comenzar, debe estar claro qué es lo que perseguimos con la automatización. ¿Buscamos *feedback* más temprano, ejecución desatendida en distintos ambientes de prueba o qué?
- **Niveles de automatización:** siguiendo el modelo de la pirámide de Cohn, deberemos pensar en qué niveles automatizar, incluyendo una buena base de pruebas unitarias, bastantes pruebas de API y solo los flujos más críticos a nivel de interfaz gráfica.
- **Enfoque de automatización** (*Scripted, Record and Playback, Model Based, Low Code*): para poder hacer una correcta selección de las herramientas que usaremos, primero debemos pensar en qué tipo de herramientas se adaptan mejor a las características y necesidades del contexto, considerando desde la tecnología hasta las habilidades del equipo.
- **CI/CD:** la estrategia de automatización debe estar alineada a la estrategia de *pipelines* de integración y entrega continua, ya que juega un papel clave.
- **Herramientas de automatización:** para cada uno de esos niveles ¿qué herramientas estaremos usando? Nuestro consejo es pensar en facilitar la colaboración con desarrollo gracias a elegir herramientas que funcionen en el mismo lenguaje y entorno de trabajo.
- **Diseño y selección de casos de prueba:** antes de comenzar a automatizar debemos tener diseñados los casos de prueba y seleccionar los que den mayor ROI para cumplir los objetivos de la automatización. Un enfoque basado en riesgos nos ayudará a priorizar, buscando maximizar la cobertura en las áreas de mayor criticidad e impacto.
- **Ambientes y datos de prueba:** ¿cómo se van a gestionar los ambientes y datos de prueba? Una mala gestión de estos aspectos podría significar mayor trabajo al identificar falsos positivos.
- **Test manual y automatizado:** es clave coordinar de manera adecuada los esfuerzos para que el test automatizado potencie el *testing*. Si las pruebas

automatizadas son confiables, buscaremos no repetir todas las mismas pruebas al ejecutar de manera manual.

- **Reportes y métricas:** algo en lo que no enfatizamos en el capítulo tiene que ver con pensar cómo vamos a mostrar el valor de las pruebas, cómo vamos a medir si nos acercamos al objetivo y a quiénes les vamos a hacer llegar esos reportes.

Comentarios finales del capítulo

“La automatización de pruebas es simplemente una forma automática de hacer lo que los *testers* estaban haciendo antes” (del inglés: *Test automation is simply an automatic way of doing what testers were doing before*, por Steve Rowe, *tester* en Microsoft).

En un principio los *testers* tenían para probar un *software*, entonces presionaban botones, invocaban funciones con distintos valores de sus parámetros y luego verificaban que el comportamiento fuera el esperado. Luego, estas funciones se volvieron más complejas, cada vez más botones, los sistemas cada vez más sofisticados y los *testers* no podían con todo. Los desarrolladores tenían que esperar mucho por el visto bueno de los *testers* para poder salir a la venta. Entonces, la solución está en el *Testing* Automatizado, que consiste en algo tan simple como un programa que ejecuta las funciones con distintos datos, toca los botones igual que una persona y luego verifica programáticamente si el resultado es el correcto. De ahí surge la frase antes citada.

Cualquier *tester* se ofendería al leer esto, ya que en realidad los *testers* no pueden ser reemplazados por máquinas! Esa frase forma parte de un *post* en un blog de Steve Rowe (2007), a lo cual James Bach le dio respuesta en su blog(2007), criticando estas afirmaciones. Entre otras cosas, destacamos esta frase que resume todo:

“La automatización no hace lo que los *testers* hacían, a menos que ignores la mayoría de las cosas que realmente hacían. El *Testing* Automatizado es para extender el alcance del trabajo de los *testers*, no para sustituirlo”.

Nosotros, desde nuestra humilde opinión, estamos más de acuerdo con James, ya que no creemos que la automatización sea capaz de sustituir el trabajo de los *testers*, sino de lograr que sea mucho mejor y de mayor alcance.

No todo se debe automatizar, y no hay que intentar reemplazar el test manual por el automático, pues hay cosas que no se pueden automatizar (más que nada si necesitan verificación visual y determinación por parte del usuario) y en ocasiones es más fácil ejecutar algo manualmente que automatizarlo. De hecho, si todas las ejecuciones se pudieran hacer en forma manual, seguramente sería mucho mejor, en el sentido de que, al ejecutarlo de esta forma, se pueden encontrar otras cosas. Porque recuerden que las pruebas automatizadas hacen chequeo y no *testing*. El tema es que lleva más tiempo hacerlo manualmente y por eso es conveniente automatizar lo que vale la pena automatizar.

Una prueba de *performance* solo al final de un proyecto es como pedir un examen de sangre cuando el paciente ya está muerto.

Scott Barber

PRUEBAS DE *PERFORMANCE*

Las pruebas de rendimiento (o pruebas de *performance*, como se les suele llamar) consisten en simular carga (múltiples usuarios conectados al mismo tiempo) para analizar el desempeño del sistema. Esto permite encontrar problemas de rendimiento, así como oportunidades de mejora. Para la simulación, se utilizan herramientas específicas, en las que se debe automatizar las acciones que generarán esa carga, esto es: las interacciones entre el usuario y el sistema que estamos probando. Para poder maximizar la cantidad de usuarios simulados minimizando los recursos necesarios (la infraestructura requerida para las pruebas), se automatizan las interacciones a nivel de protocolo, lo cual hace que la automatización sea más compleja (en cuanto al trabajo, necesario para su preparación) que la automatización de pruebas funcionales, que se realiza a nivel de interfaz gráfica. Veremos en este capítulo varios aspectos relacionados con este tipo de pruebas y el análisis de resultados, con el fin de mejorar el rendimiento de las aplicaciones.

Introducción a las pruebas de *performance*

Comencemos entendiendo qué es ***performance***. Se suele asociar a la velocidad con la que responde un sistema, a sus tiempos de respuesta, pero esto no es todo. Si un sistema responde superrápido, pero consume el 100% del CPU del servidor, su *performance* no será buena. Cuando hablamos de *performance*, nos tenemos que centrar en dos cosas: por un lado lo que el usuario percibe, que es la rapidez con la que obtiene las respuestas y, por otro, en los recursos de infraestructura que utilice (CPU, memoria, red, disco de los servidores, así como otros componentes).

Para las pruebas de rendimiento, también se utilizan herramientas de automatización, pero en este caso el objetivo es simular la carga que generan múltiples usuarios conectados concurrentemente. Estas herramientas nos permitirán medir los tiempos de respuesta, pero como recién dijimos, eso no es todo. Por eso, también se necesitarán herramientas de monitorización que nos den información del uso de la infraestructura.

Mientras se ejecuta la prueba con las herramientas de simulación de carga, se analiza el desempeño de la aplicación con las herramientas de monitorización. En este proceso, se buscan “cuellos de botella” y oportunidades de mejora. Un “**cuello de botella**” se refiere a un punto en un sistema donde la capacidad o el rendimiento se ven restringidos por no poder manejar la carga de manera eficiente. Tal como cuando se pone boca abajo una botella con líquido adentro, la velocidad con la que caiga el líquido estará restringida por el tamaño del “cuello de botella”. Identificar y resolver “cuellos de botella” es crucial para mejorar el rendimiento general de un sistema.

Con estas pruebas, podremos responder a preguntas como:

- ¿La aplicación podrá responder adecuadamente a la carga?
- ¿Cuántos usuarios podrá soportar el sistema antes de dejar de responder?
- ¿Qué tan rápido se recupera la aplicación luego de un pico de carga?
- ¿Cómo afecta a los usuarios la ejecución de determinado proceso?

- ¿Cuáles son los “cuellos de botella” del sistema?
- ¿Cómo cambia la *performance* del sistema al tener un gran volumen de datos?
- ¿Qué configuración es óptima para la operativa diaria?
- ¿Será la aplicación capaz de responder adecuadamente ese día en que tendrá mucha mayor carga que lo habitual?

Una *prueba de rendimiento* (Meier, 2007) se define como una investigación técnica para determinar o validar la velocidad, escalabilidad y/o características de estabilidad de un sistema bajo pruebas, de forma tal de analizar su desempeño en situaciones de carga.

Las pruebas de rendimiento son sumamente necesarias para reducir riesgos en la puesta en producción, lograr analizar y mejorar el rendimiento de la aplicación y de los recursos de *hardware* utilizados al exponer al sistema a usuarios concurrentes. Para ello, se utilizan herramientas específicas (llamadas *generadoras de carga*) para simular la acción de múltiples usuarios concurrentes. La herramienta generadora de carga *open source* más popular es JMeter⁴⁴. Al momento de escribir la primera versión de este libro, la que más usábamos era OpenSTA⁴⁵ pero quedó discontinuada.

Para llevar a cabo una prueba de rendimiento, generalmente se procede analizando los requisitos de *performance* y el escenario de carga al que se expondrá el sistema. Una vez que se seleccionan los casos de prueba, estos se automatizan para su simulación en concurrencia. Una vez que se tiene lista la automatización, así como la infraestructura y datos de prueba y la monitorización, se pasa a ejecutar y analizar los resultados. Si bien vamos a ver muchos puntos importantes de estas pruebas, recomendamos una serie de artículos publicados por Scott Barber (2001) llamados *User experience, not metrics*⁴⁶.

⁴⁴ JMeter: <<http://jmeter.apache.org>>

⁴⁵ OpenSTA: <<http://opensta.org>>

⁴⁶ Para profundizar según Scott Barber: *User Experience, not Metrics*:
<<http://www.perftestplus.com/pubs.htm>>

Tipos de pruebas de *performance*

Hay distintos tipos de pruebas de *performance*, cada una intenta responder a distintos tipos de preguntas o incertidumbres que podemos plantear, las cuales representan riesgos para la puesta en producción de nuestro sistema:

- **Prueba de estrés (*stress testing*):** Se busca determinar la cantidad máxima de usuarios concurrentes que soporta el sistema con una experiencia de usuario aceptable, identificando así lo que se conoce como el punto de quiebre.
- **Prueba de carga (*load test*):** Se busca simular el escenario esperado de carga del sistema para entender cómo se comportará la aplicación y qué oportunidades de mejora se encuentran para ese escenario esperado.
- **Prueba de *endurance*:** El objetivo es entender cómo funcionará el sistema luego de estar cierto tiempo ejecutando, por ejemplo, luego de estar un día entero atendiendo las solicitudes de los usuarios.
- **Prueba de picos:** Se busca simular situaciones de carga puntuales conocidas como *picos*, que suelen ser cargas mayores por períodos cortos (la casuística hace que en un mismo momento coincidan muchas más peticiones que lo normal, ante lo cual sabemos que los tiempos de respuesta quizá empeoran por debajo de lo aceptable). Lo que se busca estudiar es si el sistema logra recuperarse y qué tan rápido lo logre luego de esta exigencia fuera de lo normal.
- **Ingeniería del caos:** Son pruebas llevadas a cabo en sistemas de alcance global y de alta demanda, donde en producción y de manera intencional se afectan componentes de la infraestructura para entender qué tan resiliente es el sistema. Estas pruebas se popularizaron de la mano de Netflix y su plataforma *open source* para esto, llamada *Chaos Monkey*⁴⁷.

Según cuáles sean los riesgos que queremos mitigar, será el tipo de prueba que tendremos que planificar hacer.

⁴⁷ *Chaos Monkey*: <<https://netflix.github.io/chaosmonkey>>

Introducción al proceso de pruebas de *performance*

Si bien en los últimos años hemos adoptado más prácticas ágiles para enfocar el *Testing de Performance*, en este capítulo estaremos mostrando más que nada un enfoque más tradicional, en etapas, ya que es el caso más común en el que se llevan adelante.

Cualquiera que sea el tipo de prueba de *performance* que se esté haciendo, hay toda una preparación necesaria que requiere un esfuerzo no despreciable.

1. En primer lugar, es necesario hacer un análisis y diseño de la carga que se quiere simular (duración de la simulación, casos de prueba, las cantidades de usuarios que los ejecutarán y cuántas veces por hora, *ramp-up*⁴⁸, etc.).
2. Luego de diseñado el “escenario de carga” pasamos a preparar la simulación con la/s herramienta/s de simulación de carga, lo cual es, en cierta forma, una tarea de programación en una herramienta específica (existen muchas, por ejemplo, las opciones *open source* que más usamos hoy son JMeter, Taurus⁴⁹ y Gatling⁵⁰). A esta tarea se la suele denominar como *automatización o robotización*.
3. Por último, comenzamos con la ejecución de las pruebas. Esto lo hacemos con toda la simulación preparada y las herramientas de monitorización configuradas para obtener los datos de desempeño de los distintos elementos que conforman la infraestructura del sistema bajo pruebas. Esta etapa es un ciclo de ejecución, análisis y ajuste, que tiene como finalidad lograr mejoras en el sistema con los datos que se obtienen luego de cada ejecución.

Ahora, ¿cuándo las llevamos a cabo? Hemos visto muchas veces que no se tienen en cuenta temas relacionados con el rendimiento del sistema cuando está en producción. También pasa que se subestima la importancia y el costo de las pruebas. ¿Quién no ha oído estas frases al hablar de pruebas de *performance*?

⁴⁸ *Ramp-up*: Velocidad y cadencia con la que ingresan los usuarios simulados durante una prueba.

⁴⁹ Taurus: <<https://gettaurus.org>>

⁵⁰ Gatling: <<https://gatling.io>>

- “Lo dejamos para el final, solo si hay tiempo”.
- “Le pedimos a alguno de los muchachos que tenga un rato libre que vea cómo usar estas herramientas y ejecute una pruebita”.
- “Unos días antes de salir en producción contratamos alguna empresa que nos dé servicios y vemos que todo vaya bien”.

Sí, es cierto que estas son opciones, pero también es cierto que son muy arriesgadas. Como a tantas otras cosas, hay que dedicarle tiempo a estos aspectos si son requeridos. Quizá haya casos en los que la *performance* no requiera tanto esfuerzo para evaluar (un sistema que sea monousuario, que sea accedido por dos o tres personas), pero generalmente no es fácil de probar y debemos planificar el esfuerzo, no solo de la prueba, sino de la corrección de los incidentes que se van a encontrar.

Como dice Scott Barber, si dejamos las pruebas para el final, será como “pedir un examen de sangre cuando el paciente ya esté muerto”. Las medidas correctivas serán mucho más caras de implementar. Es tomar un riesgo muy alto.

Sin embargo, ¿cuánto nos podemos anticipar? Si el sistema aún no está estable, no se han pasado las pruebas funcionales, por ejemplo, y nos proponemos ejecutar la prueba de *performance* del sistema, entonces nos arriesgamos a:

1. Que nos encontremos con los errores funcionales mientras automatizamos o ejecutamos las pruebas de *performance*.
2. Que ajustemos el sistema para que su *performance* sea buena y, luego, en las pruebas funcionales, se detecten problemas que impliquen cambios, y no sabemos cómo estos cambios impactarán la *performance* final.

Entonces, lo mejor sería ejecutar las pruebas completas al final del proyecto, simulando toda la carga que deberá soportar, pero también ir ejecutando pruebas intermedias paralelas con el desarrollo. Se podría ejecutar pruebas anticipadas sobre un módulo o funcionalidad específica, apenas se cuente con una versión estable de esta; el propio desarrollador podría ejecutar sus propias pruebas de *performance* en menor escala, etc. Así, se llega a las pruebas finales con una mejor calidad, y se reduce la probabilidad de que haya que realizar grandes cambios.

En el caso de sistemas con arquitecturas en capas, donde se pueda probar la capa de servicios (la API), se recomienda probar a ese nivel e, incluso, considerar agregar pruebas tempranas y que se repitan frecuentemente al proceso de desarrollo (por ejemplo, agregando las pruebas al esquema de integración continua).

A continuación, veremos distintas secciones siguiendo en forma cronológica el transcurso de un proyecto típico de una prueba de *performance*, que comienza con su diseño, preparación (automatización), ejecución y análisis de resultados.

Diseño de pruebas de *performance*

Para ver cómo definir una prueba de *performance*, iremos viendo los distintos puntos que será necesario determinar, siempre apuntando a obtener resultados oportunos que nos permitan reducir riesgos. Si nos enfocamos a hacer una prueba perfecta, seguramente vamos a gastar muchos recursos y dinero y los resultados pueden llegar tarde.

Veremos entonces que, para diseñar una prueba de *performance*, tendremos que definir el alcance y objetivos de las pruebas, los escenarios de carga (los casos de prueba y la mezcla de usuarios que queremos ejecutar), los datos de prueba, la infraestructura de pruebas y cómo vamos a medir los resultados, además de cuáles serán nuestros criterios de aceptación.

Definición de alcance, objetivos y criterio de finalización de prueba

¿Por qué se quiere ejecutar pruebas de *performance*? Se pueden encontrar distintos motivos y, de acuerdo a cuáles son los objetivos, se determinarán muchos factores decisivos en el diseño de la prueba, por lo que esto es lo primero que vamos a definir.

Por mencionar los más comunes, podríamos listar:

- **Salida a producción por primera vez:** hemos trabajado mucho tiempo en construir el sistema y por primera vez lo dejaremos accesible al mercado, al público objetivo, y acá las primeras impresiones pueden ser fundamentales para el éxito del negocio.
- **Cambios en la infraestructura.** Quizá nuestro sistema ya está funcionando en determinada plataforma y por algún motivo queremos cambiar la plataforma, pasar a otro equipo más potente (o más económico), cambiar un componente o reestructurar la red interna de la empresa.

- **Próxima liberación de una nueva versión del sistema.** Puede tratarse de nuevas funcionalidades o cambios en funcionalidades existentes.
- **Cambio de arquitectura.** Por ejemplo, al pasar de arquitectura cliente-servidor a plataforma *full-web* y ahora incluyendo un módulo en *mobile*.
- **Ajustes de configuración.** Se quiere probar una nueva configuración de la base de datos, de los servidores de aplicaciones o de cualquier componente de la infraestructura.
- **Aumento de la carga esperada.** Quizá se está lanzando una nueva estrategia de *marketing* y gracias a ella se prevé que la cantidad de usuarios que va a acceder al sistema aumentará considerablemente. Esto puede ser también pensando en una fecha especial, tal como Navidad, *Black Friday*, fin de mes, un Mundial de fútbol o cualquier acontecimiento que aumente el pronóstico de uso del sistema.

Pueden existir muchos motivos, al tenerlos claros podremos determinar qué cosas están dentro de nuestro alcance y también así sabremos cuándo vamos a dar por concluidas las pruebas, una vez que hayamos verificado la validez de nuestro nuevo escenario.

Selección de las herramientas

Sin entrar en demasiados detalles o en todas las opciones disponibles hoy en día, nos parece importante mencionar cuál es el enfoque moderno de las herramientas de pruebas de simulación de carga para tenerlas presente y así poder seleccionar la que mejor se adapte a las necesidades, prestando atención a los costos asociados también.

La mayoría de las herramientas hoy en día se basan en un producto *open source* para preparar las pruebas y ejecutar una cantidad acotada. Generalmente, JMeter, Gatling y otras nos van a permitir ejecutar entre 500 y 1000 usuarios simultáneamente desde una PC convencional. La mayoría de las veces vamos a

necesitar simular muchos más usuarios que eso, por lo cual la mejor estrategia es escalar en la nube.

Es así como herramientas como Blazemeter⁵¹, K6⁵², Gatling, Octoperf⁵³ y otras nos brindan la posibilidad de subir el *script* a la nube y ejecutarlo con muchos más usuarios (incluso cientos de miles). Además de la escalabilidad de la prueba que brindan, nos dan más funcionalidades que son muy interesantes:

- Ejecución distribuida, se puede ejecutar la prueba desde distintos puntos del planeta.
- Simulación de anchos de banda, para tener en cuenta que algunos de nuestros usuarios estarán accediendo al sistema con diferentes tipos de conexiones, por ejemplo 3G o con muchos errores por problemas de señal.
- Reportes integrados, recolectando fácilmente los datos de las ejecuciones de todas las generadoras de carga que estarán distribuidas e, incluso, brindando la posibilidad de cruzar esa información con la monitorización que tengamos disponible.

Estas herramientas suelen cobrar por una combinación de cantidad de usuarios que se simulan y horas de simulación. Incluso existe el concepto de *virtual user hour* (*horas de usuario virtual*). Es importante desde ya definir qué cantidad de carga estaremos ejecutando y por cuánto tiempo como para poder estimar el costo de estas herramientas y elegir la que más se ajuste a nuestras necesidades.

Casos de prueba

Comencemos hablando de qué casos de prueba incluiremos en nuestras pruebas. En este tipo de pruebas es muy distinto a como uno piensa y diseña los casos de prueba funcionales. Tenemos otros criterios de cobertura, otros criterios de riesgos como para decidir qué probar y qué dejar afuera. Hay que tener presente que aquí la automatización es obligatoria y es cara, entonces cada caso de prueba que

⁵¹ Blazemeter: <www.blazemeter.com>

⁵² K6: <k6.io>

⁵³ Octoperf: <www.octoperf.com>

incluimos implicará más trabajo de automatización, preparación de datos, mantenimiento de la prueba, etc. Entonces, tendremos que limitar la cantidad de casos de prueba a simular y esto hace que sea muy importante seleccionarlos adecuadamente. En nuestra experiencia, dependiendo de las dificultades técnicas de la aplicación para su automatización y del tiempo disponible para la preparación, intentamos no pasar de 15 casos de prueba.

Los criterios de cobertura que tenemos para seleccionar los casos de prueba se basan en seleccionar los casos de acuerdo a:

- **Cuáles son las funcionalidades más intensivas en uso de recursos.** Es fundamental preguntar a los responsables de infraestructura e incluso a los desarrolladores, pues ellos tendrán idea de cómo programaron cada cosa.
- **Cuáles son los más riesgosos para la operativa del negocio.** Es fundamental preguntarle al que pierde dinero si algo no funciona. También a los usuarios.
- **Cuáles son los más usados.** Es fundamental mirar *logs*, preguntar a los usuarios y expertos funcionales.

Cada caso de prueba tendrá que ser diseñado con cuidado. Tenemos que definir un guion, indicando cuáles son los pasos para poder ejecutarlo, qué parámetros tiene (qué datos de entrada) y cuál es la respuesta esperada, o sea, qué vamos a validar tras cada ejecución.

Hay que tener presente que las validaciones en estas pruebas no serán iguales a las validaciones en las pruebas funcionales. No es el objetivo verificar que los cálculos se hicieron correctamente, sino que la idea es verificar mínimamente que la operación se procesó. Entonces, las verificaciones generalmente se limitan a verificar que aparezca en pantalla un mensaje diciendo “transacción procesada con éxito”, que no aparece un mensaje de error o que aparece el nombre del usuario que hizo *login*. Validaciones muy extensivas o detalladas implicarán una mayor utilización de recursos de *hardware* en la máquina que ejecute estas pruebas, por lo que este es un aspecto del cual también debemos tener cuidado.

Por otro lado, y es algo que distingue mucho a un caso de prueba funcional de uno de estas pruebas, es que aquí el tiempo es importante. Vamos a querer simular cada caso de prueba en los tiempos en que un usuario lo ejecutaría. No está bien ejecutar

una acción tras la otra, pues un usuario real no lo hace así. Generalmente invoca una funcionalidad, luego recibe una pantalla y la lee, decide qué hacer después, ingresa datos en un formulario, etc. Entonces, la estrategia es agregar pausas en el *script*, de forma tal que se simulen esos tiempos de usuarios a los que se les llama *think time*. Si ejecutamos una simulación sin considerar los *think times*, probablemente la prueba nos dé falsos positivos, o sea, nos hará investigar errores o veremos necesario implementar optimizaciones en un sistema que exigimos más de lo que estará expuesto en producción.

No todos los usuarios ejecutan las funcionalidades de la misma forma. Generalmente podrán seguir distintos flujos, elegir distintos parámetros, distintas opciones, ejecutar con distinta velocidad (imaginen que hay vendedores que ingresan un pedido cada 10 minutos porque van puerta a puerta y otros por teléfono ingresan uno por minuto), etc. Intentar cubrir todas las opciones también puede generar un costo muy alto. Acá podemos seguir distintas estrategias. Una es promediar el comportamiento, o sea, considerar el comportamiento más común y hacer que todos los usuarios sigan ese mismo comportamiento. Otra opción es generar un par de grupos, como por ejemplo, si consideramos las diferencias en el tiempo de ejecución de cada usuario, podríamos tener el grupo de los usuarios “liebre” y el grupo de los usuarios “tortuga”. Se trata del mismo caso de prueba, pero uno va a estar configurado para que se ejecute con menos pausas que el otro o con mayor frecuencia que el otro.

Otro tipo de simplificación que puede hacerse, a modo de mejorar la relación costo/beneficio de la prueba, es hacer jugar un caso de prueba por otro. O sea, imaginemos que analizamos el uso del sistema y se llega a que hay 200 usuarios que realizan extracción por cajero y unos 20 que realizan pagos con tarjeta en comercio. Para el sistema, las dos operaciones usan la misma interfaz y son muy similares. Entonces, podríamos considerar que tenemos 220 usuarios ejecutando extracciones. Esto nos reduce el costo a la mitad, ya que solo tenemos que preparar un *script* y nos da un beneficio similar a tener los dos *scripts*. Entonces, la prueba no será exacta ni perfecta, pero será en buena medida cercana a la realidad. No decimos que esto haya que hacerlo siempre, sino cuando no hay tiempo ni recursos para preparar las dos pruebas.

Escenarios de carga

Los casos de prueba no se ejecutarán secuencialmente como suele hacerse en las pruebas funcionales. En una prueba de *performance* simulamos la carga de trabajo que tendrá una aplicación cuando esté en producción, contando los usuarios concurrentes que estarán accediendo, los casos de prueba que serán ejecutados, la frecuencia con la que cada usuario ejecuta, etc., etc., etc. Para esta definición, es necesario contar con conocimiento del negocio, saber cómo se van a comportar los usuarios, qué van a hacer, etc.

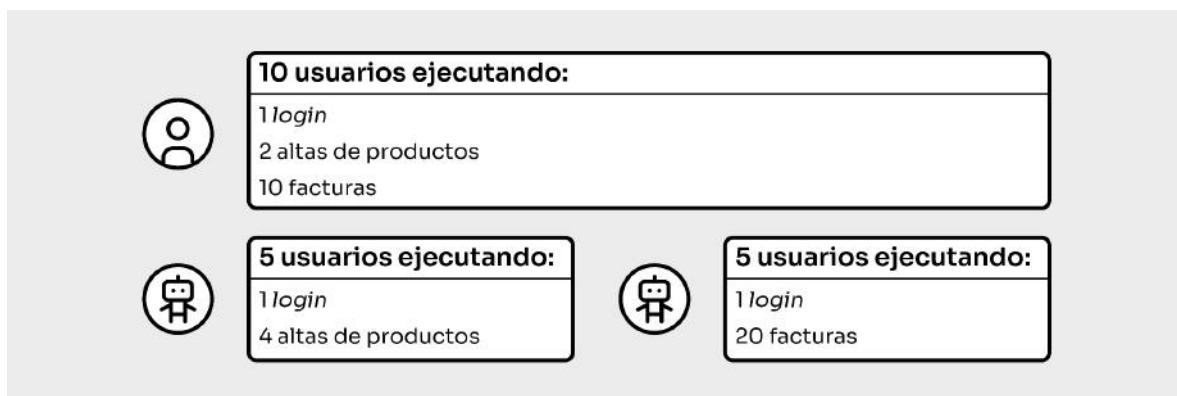
Podríamos definir así el concepto de **workload** o *escenario de carga* como ese conjunto de características que describen la carga esperada del sistema en un momento determinado (en la operativa diaria durante la hora pico, en un día de mayor uso del sistema, etc.).

Cuando preparamos una prueba de *performance*, diseñamos un *workload* considerando ciertas **simplificaciones** para así poder mejorar la relación costo/beneficio. O sea, si pretendemos realizar una simulación idéntica a la que el sistema recibirá cuando esté en producción, vamos a terminar haciendo una prueba tan pero tan cara que no valdrá la pena: quizá los beneficios sean menores que los costos o quizá ilos resultados lleguen demasiado tarde! ¿Ya dijimos esto? Sí, es cierto, pero mejor repetirlo.

Generalmente se piensa en simular **una hora pico** del sistema. O sea, la hora en que el sistema tiene mayor carga, el día de la semana de mayor carga, en el mes de mayor carga, etc. Se puede querer simular distintos escenarios (quizá en la noche se ejecuten algunas funcionalidades con distinta proporción/intensidad que en el día, entonces tendremos un escenario diurno y otro nocturno). Lo importante, y lo que nosotros generalmente aplicamos en la práctica, es **definir el escenario en base a una hora de ejecución**. En nuestra experiencia, la duración de una hora es adecuada en costo y beneficio porque es suficientemente larga para poner al sistema “en régimen” para observar “cuellos de botella”, y suficientemente corta para agilizar el proceso y repetir varias pruebas en un mismo día.

Otro concepto muy importante para entender cómo se modela un escenario de carga es el de **especialización de usuarios**. Generalmente, los usuarios reales

ejecutan tareas variadas, acceden al sistema, ingresan datos, consultan, etc. En la simulación, vamos a especializar los usuarios, o sea, habrá usuarios que solo ingresen datos y habrá otros que solo consulten datos. Dicho de otra forma, en la simulación, un grupo de usuarios virtuales ejecutará una y otra vez un caso de prueba automatizado y otro grupo de usuarios virtuales ejecutará otro. Lo importante es diseñar correctamente la mezcla de usuarios y ejecuciones, de forma tal que para el sistema sea lo mismo la carga total que reciba por parte de los usuarios reales como por parte de los usuarios virtuales. En la siguiente figura, se muestra un ejemplo de un escenario real y un escenario simulado que deberían ser equivalentes para el punto de vista del sistema.



Escenarios equivalentes de usuarios reales y usuarios virtuales

Podríamos decir que el comportamiento de los usuarios suele ser muy estocástico, lo cual hace que sea muy complejo y, de ahí, que necesitamos simplificarlo. Para aclarar, un proceso estocástico es un proceso aleatorio en el que las variables cambian de manera probabilística en lugar de seguir un patrón determinista.

Creemos que es muy interesante hacer un análisis y ver por qué típicamente en las herramientas de simulación de carga se busca simplificar la forma en la que se definen los escenarios de carga:

- **Queremos pruebas repetibles.** Nunca una ejecución puede ser exactamente igual a otra, pero se busca la repetibilidad de la prueba. Por ejemplo, si ejecutamos esta prueba, encontramos una oportunidad de mejora y luego queremos ver si el resultado tras ese cambio fue para mejor o no, qué tanto mejora la *performance*, etc., deberíamos intentar ejecutar la

misma prueba. Si ejecutamos una prueba que tiene un factor variable (basado en probabilidades), estamos perdiendo esta posibilidad.

- **Queremos mejorar la relación costo-beneficio**, simplificando la realidad sin perder realismo, pero sin exagerar en costos. ¿Cuánto más nos costaría definir el modelo matemático y asegurarnos que sea válido?
- **Queremos que el cliente sea capaz de definir la prueba y comprenderla**. Es difícil que alguien (con conocimiento del negocio, de la aplicación, etc.) haga estimaciones sobre las cantidades de facturas que se crearán en una hora pico, mucho más si comenzamos a hablar de conceptos matemáticos, con lo cual esto agregaría complejidad a la hora de definir y diseñar una prueba. Si logramos que esto sea simple de comprender, lograremos que alguien con conocimiento del negocio pueda entender qué cosas cambiar de la lógica para poder solventar los problemas.

Creemos que la simulación estocástica sirve para algunos sistemas en donde las variables, entornos y comportamientos son conocidos y estables (a nivel de *hardware*, *drivers*, protocolos, etc.). Pero estas variables solamente pueden obtenerse realizando simulaciones de carga reales. Lograr definir todas las variables y su comportamiento en correlación con las otras es una tarea muy compleja (¿se necesitaría poder simular matemáticamente además el comportamiento de los usuarios y sus efectos sobre el entorno?).

La prueba de simulación de carga de usuarios nos permite no solo tener un set de pruebas que simulan el comportamiento de un usuario, sino que me permite además mezclar y crear diferentes escenarios y probarlos ayudando a eliminar la incertidumbre y a lograr detectar posibles fallos frente a casos de concurrencia no pensados.

Ejemplos:

- ¿Qué sucedería si mientras ejecutamos determinado *workload* borramos las sesiones del sistema? ¿Cómo se comportaría?
- ¿Qué sucede si bloqueamos un determinado registro de la base de datos?
- ¿Qué sucede si simulamos un problema de red?
- ¿Qué sucede si simulamos una caída de uno de los balanceadores?

Teniendo estos puntos en consideración, llegamos a la conclusión de que este tipo de enfoques (sin aplicar procesos estocásticos al modelado del escenario), para este tipo de pruebas, es mejor en cuanto a costo/beneficio.

Infraestructura y datos de prueba

La infraestructura y datos de prueba son dos puntos a determinar muy distintos, pero los colocamos juntos por un motivo muy importante: tienen que ser ambos lo más parecido a lo que tendremos cuando el sistema esté en producción. Si probamos en una infraestructura diferente, **no podremos extrapolar los resultados** a la infraestructura de producción. Esto es, que un ambiente tenga el doble de recursos que el otro no significa que vaya a responder el doble de rápido, o que soporte el doble de carga. Si utilizamos la base de datos de pruebas, seguramente no detectemos las consultas SQL que sean poco eficientes, ya que muchas veces estas bases de datos tendrán menos registros o no tendrán la misma distribución que en producción, con lo cual ineficiencia de las consultas puede pasar inadvertida.

Entonces, intentaremos usar los servidores de producción (si no están en uso) e intentaremos usar la base de datos de producción. Si esto fuera posible, habría que tomar muchos recaudos para no generar problemas, pero eso dependerá caso a caso. Si hay restricciones de su uso, ya sea por confidencialidad de datos o por disponibilidad del ambiente o por el riesgo de afectar a los usuarios en producción, entonces, tendremos que analizar la factibilidad de enmascarar esos datos, o de generar una base de datos de prueba artificial, pero con un volumen y distribución similar. La validez de los resultados dependerá en gran parte de estos factores.

Herramientas e indicadores para la monitorización

Es importante determinar cuál es la infraestructura bajo pruebas, qué componentes la integran, pues eso será lo que determinará qué componentes estaremos monitorizando y analizando. Por ejemplo, el servidor de correos es parte

de la infraestructura de nuestra empresa, pero seguramente no esté tan vinculado a la aplicación bajo pruebas, con lo cual no nos interesará incluirlo en la infraestructura de pruebas. Una vez que definimos qué elementos la integran, sabremos qué componentes monitorizar y qué indicadores recogeremos, con qué herramientas y cuáles son los umbrales aceptables.

En la ejecución y monitorización es donde más hace falta el conocimiento detallado de los componentes del sistema. Con *componentes*, nos referimos tanto a los componentes internos como al *software* de base que utilizamos: servidor *web*, servidor de base de datos, sistemas operativos, etc.

En esta selección de herramientas e indicadores es importante también pensar en lo que tendremos en producción. Las herramientas de monitorización también consumen recursos, los mismos recursos que estarán midiendo. Entonces, si en producción tendremos ciertas herramientas para verificar la salud de los servidores, sería ideal utilizar esas mismas herramientas para analizar la infraestructura durante las pruebas. Esto tiene muchas ventajas:

- Se van familiarizando los integrantes del equipo con las herramientas que utilizarán en producción para controlar la infraestructura.
- Se ajustan las herramientas para su óptimo rendimiento y para obtener toda la información útil.
- Se asegura que las herramientas no entran en conflicto entre ellas o con el sistema bajo pruebas.
- Se verifica qué tan intrusivas son estas herramientas, o sea, qué tanto impacta la *performance* la propia herramienta de análisis de *performance* al estar tomando mediciones.

Además, una estrategia muy buena para la monitorización es la de dividir en dos grupos de indicadores. Primero, tendremos los **indicadores de primer nivel**, que serán los que estaremos verificando siempre y serán los que estaremos recogiendo en producción, ya que se pueden medir con herramientas no intrusivas; por otro lado, tendremos los **indicadores de segundo nivel**, cuyo objetivo es obtener más información con más detalle sobre un componente y son los que generalmente activaremos cuando ya detectamos un problema y queremos saber sus raíces. El problema de los indicadores de segundo nivel es que generalmente

son más intrusivos y, por eso, no podemos mantenerlos siempre encendidos, como por ejemplo un *trace* a nivel de base de datos. A modo de ejemplo, en primer nivel tendremos el CPU, memoria, red y disco de cada componente, y en segundo nivel podrán incluirse logs con más detalles.

Todo lo expuesto en esta sección de monitorización hoy en día queda mucho más simple y con mucho más poder si utilizamos herramientas de observabilidad⁵⁴.

Diseño de oráculos y criterios de aceptación

¿Cuándo consideramos que un resultado es aceptable? Hay quienes dicen que hay una regla que indica que un usuario en un sitio *web* no espera más de 8 segundos. Esto se decía antes de la fibra óptica y la banda ancha, quizá si queremos seguir pensando en esa regla la tendríamos que bajar a 3 segundos o menos. De hecho, hay un estudio de Google en 2016 que plantea que el 53% de los usuarios que visitan un sitio desde un dispositivo móvil lo abandonan si demora más de 3 segundos en cargar. Pero ¿es tan así? ¿Y si lo que el usuario está haciendo es la declaración de la renta, impuestos, obteniendo un listado que realmente necesita para su trabajo o algo similar? Seguramente, en esos casos sea un poco más paciente y espere por la respuesta del sistema.

El oráculo en una prueba de *performance* se define en forma global. En lugar de mirar los resultados individuales de cada prueba, se miran en su totalidad y se analizan en forma agregada. Esto es, que se le prestará atención los tiempos de respuesta de forma agregada, por ejemplo, el 90% de los usuarios tuvieron tiempos de respuesta menores a 2,3 segundos. De igual manera, se mirará el porcentaje de fallos que se obtuvo, y no a cada una de las ejecuciones.

Hay quienes dicen que los proyectos generalmente fallan por problemas de definición de requisitos y que esto se aplica a las pruebas de *performance* también. Entonces, es sumamente importante para el éxito de una prueba de *performance* que establezcamos los valores aceptables de tiempos de respuesta (u otras medidas). O sea, antes de comenzar a ejecutar ninguna prueba, en las primeras

⁵⁴ Información sobre *observabilidad*: <[https://en.wikipedia.org/wiki/Observability_\(software\)](https://en.wikipedia.org/wiki/Observability_(software))>

etapas del proyecto, hay que establecer cuántos segundos es el máximo aceptable para la respuesta de una página o funcionalidad.

Desde nuestro punto de vista, en la práctica esto es casi imposible. Al menos en nuestra experiencia casi nunca ha podido hacerse en forma tan exacta. De todos modos, veremos alguna alternativa.

Lo que vemos siempre complicado con respecto a esto son dos cosas:

- Encontrar a alguien que *se la juegue* a dar un número de tiempo de respuesta aceptable. Generalmente, no se tiene idea, nadie fija un número o incluso se ve como algo más subjetivo que numérico.
- Por otro lado, si definimos que lo aceptable es 5 segundos (por decir algo) y al ejecutar las pruebas nos da 7 segundos: ¿entonces la prueba está diciendo que el sistema está mal? Tal vez, al interactuar con el sistema, entrando a la página que tarda 7 segundos, al primer segundo ya tenemos el formulario que hay que completar en el siguiente paso y el resto de los 7 segundos la página termina de cargar sus imágenes y otras cosas, pero funcionalmente es más que aceptable el tiempo de respuesta que presenta.

Entonces, basados en estas dudas, proponemos cambiar el foco y, en lugar de plantear el objetivo de definir lo aceptable o no de una prueba en base al tiempo de respuesta, hacerlo en base a las necesidades del negocio: **la cantidad de datos a procesar por hora.**

Por ejemplo:

- Se necesita procesar 2500 facturas en una hora.
- En las cajas se atienden 2 clientes por minuto en cada una.

En base a esta definición es posible determinar si se está cumpliendo con esas **exigencias del negocio** o no. Esto, además, se enriquecería con la visión subjetiva de un usuario: mientras se ejecuta/simula la carga de usuarios concurrentes sobre el sistema, un usuario real (o usuario testigo) puede entrar y comprobar la velocidad de respuesta del sistema, viendo si está usable o insoportablemente lento. Por otro lado, como en todas las pruebas de *performance*, estos resultados hay que correlacionarlos con los consumos de recursos. Pueden existir necesidades de no sobrepasar determinados umbrales, como por ejemplo que el CPU del servidor de base de datos tenga siempre un 30%

libre, pues se necesita para procesar alguna otra cosa, etc. De hecho, también hay recomendaciones para esto basadas en distintos estudios, brindados en general por los proveedores. Por ejemplo, IBM recomienda que el CPU de los servidores se encuentre entre el 70% y el 80% de uso.

Un concepto explicado por Harry Robinson (el precursor del *Model-Based Testing* que nombramos antes) es el de usar **oráculos basados en heurísticas**, o sea, pensar en cosas que sean *más o menos* probables, por más que no tengamos la total certeza del resultado esperado. Harry plantea un ejemplo muy bueno de cómo probar Google Maps, indicando que se podría probar calcular el camino de ida de *A* hacia *B*, luego de *B* hacia *A*, comparar las distancias, y si da una diferencia mayor de cierto umbral entonces marcarlo como posible error. Eso es una estrategia interesante para considerar en el diseño de oráculos, pensar en este tipo de “heurísticas” o de “oráculos probables”.

Se puede aplicar un enfoque similar en pruebas de *performance*, cuando se procesan grandes volúmenes de datos. Imaginen, por ejemplo, que tenemos una base de datos inicial con 200 clientes y estamos ejecutando casos de prueba que crean clientes. No podemos verificar que todos los datos hayan sido creados correctamente porque sería muy costoso (más allá de las verificaciones que ya hacen los casos de prueba), pero sí podemos hacer una verificación al estilo de las que planteaba Harry, en la que podemos ejecutar un *Select* para ver cuántos registros hay en determinadas tablas antes y después de la ejecución de la prueba. ¿Cómo podemos estimar más o menos cuántos clientes se crean en la prueba? Por ejemplo, asumiendo que tenemos una prueba de *performance* de 400 usuarios concurrentes creando 10 clientes por hora, y la prueba dura 30 minutos, nos da un total de 2000 clientes nuevos. Entonces, podríamos verificar que la diferencia entre el estado inicial y final sea aproximadamente de 2000 registros. Este oráculo no es preciso, pero al menos nos da una idea de qué tan bien o mal resultó.

Preparación de pruebas de *performance*

A esta altura, ya sabemos la prueba que queremos simular, ahora tenemos que prepararla. Esto implica varias actividades, algunas de infraestructura, como preparar el sistema bajo pruebas y la base de datos con los datos definidos, etc., y, por otra parte, la automatización de los casos de prueba y la configuración de los escenarios de carga, así como también la configuración de las herramientas de monitorización.

Automatización de casos de prueba

A diferencia de los *scripts* de pruebas funcionales, en estos *scripts*, si bien se utiliza el enfoque *Record and Playback*, no se graba a nivel de interfaz gráfica, sino que a nivel de protocolo de comunicación. Esto es porque en una prueba funcional, al reproducir, se ejecuta el navegador y se simulan las acciones del usuario sobre este. En una prueba de rendimiento se van a simular múltiples usuarios desde una misma máquina, por lo que no es factible abrir gran cantidad de navegadores y simular las acciones sobre ellos, ya que la máquina utilizada para generar la carga tendría problemas de rendimiento y obtendría así una prueba inválida. Al hacerlo a nivel de protocolo, se puede decir que se “ahorran recursos”, ya que en el caso del protocolo HTTP lo que tendremos serán múltiples hilos que enviarán y recibirán texto por una conexión de red y no tendrán que desplegar elementos gráficos ni ninguna otra cosa que exija mayor procesamiento.

Cabe destacar que, si bien desde la generadora de carga no se ve el navegador ejecutando el caso de prueba como en las pruebas funcionales, desde el punto de vista del servidor no hay forma de diferenciar la ejecución de un usuario real de la de un *script* de *performance* automatizado.

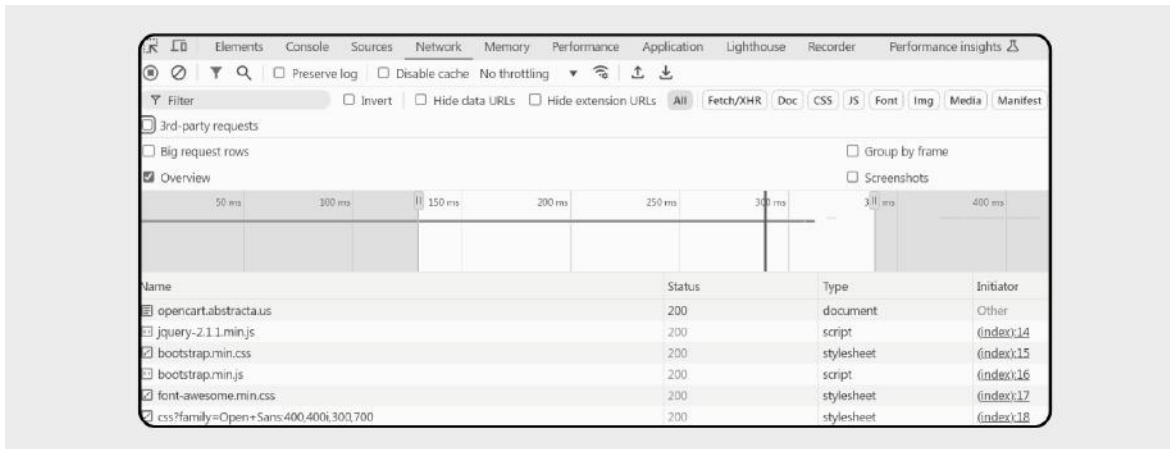
Para preparar un *script*, se procede en forma similar a lo explicado para los *scripts* de pruebas funcionales, pero esta vez la herramienta, en lugar de capturar las

interacciones entre el usuario y el navegador, capturará los flujos de tráfico HTTP entre el navegador y el servidor. Entonces, para la automatización se necesitan conocimientos sobre herramientas de automatización y protocolos de comunicación (por nombrar algunos otros protocolos de comunicación, podemos mencionar SIP, SOAP, ISO8583, HLS, WebSocket, etc).

El *script* resultante será una secuencia de comandos en un lenguaje proporcionado por la herramienta utilizada, en el cual se manejen los *requests* y *responses* de acuerdo al protocolo de comunicación. Esto da un *script* mucho más complejo de manejar que uno para pruebas funcionales. Por ejemplo, con un *script* a nivel de Selenium se puede simular la ejecución de una acción de un usuario con un comando, pero esto se puede llegar a corresponder con cientos de comandos a nivel de protocolo. Esto corresponde a cada uno de los *HTTP Requests* enviados al servidor, uno para acceder a la página inicial, uno para acceder al menú de búsqueda y uno para hacer la búsqueda. Pero, además, cada *request* desencadena una serie de *requests* secundarios extra, correspondiente a las imágenes utilizadas en la página *web*, archivos de estilos CSS, Javascript y otros recursos. Incluso pueden desencadenarse redirecciones de un objeto a otro, lo cual implica realizar más *HTTP Requests* (en respuesta a un *HTTP Request* de código *302-redirect*).

Cada *request* (primario o secundario) se compone de un encabezado y un cuerpo de mensaje. Embebidos, viajan parámetros, *cookies*, variables de sesión y todo tipo de elementos que se utilicen en la comunicación con el servidor.

A modo de ejemplo y de entender esto mucho más, les recomendamos acceder a una página *web* desde el navegador, por ejemplo desde Google Chrome, pulsar botón derecho sobre cualquier lugar de la página, elegir “Inspeccionar elemento” y ahí ir a la pestaña de red (*Network*). Entonces se podrán ver los *requests* enviados al servidor, como, por ejemplo, se puede ver en la siguiente imagen.



DevTools de Google Chrome para capturar información de tráfico HTTP

Una vez que se graba el *script*, es necesario realizar una serie de ajustes sobre estos elementos para que quede reproducible. Estos *scripts* serán ejecutados por usuarios concurrentes, y, por ejemplo, no tiene sentido que todos los usuarios utilicen el mismo nombre de usuario y clave para conectarse o que todos los usuarios hagan la misma búsqueda (ya que en ese caso la aplicación funcionaría mejor que de usar diferentes valores, dado que influirían los cachés, tanto a nivel de base de datos como a nivel de servidor de aplicaciones *web*). El costo de este tipo de ajustes dependerá de la herramienta utilizada y de la aplicación bajo pruebas. En ocasiones es necesario ajustar *cookies* o variables, pues las obtenidas al grabar dejan de ser válidas, deben ser únicas por usuario. Habrá que ajustar parámetros, tanto del encabezado como del cuerpo del mensaje, etc.

En base a nuestra experiencia ejecutando ya cientos de pruebas de rendimiento (comenzamos en el 2005 con nuestros primeros *scripts* y seguimos hasta hoy), la construcción de estos *scripts* ocupa entre el 30% y el 50% del total del esfuerzo invertido en las pruebas de rendimiento. Por otra parte, el mantenimiento de estos *scripts* suele ser tan complejo que muchas veces se prefiere rehacer un *script* de cero en lugar de ajustarlo. De esta forma, el proceso se vuelve poco flexible en la práctica. Las pruebas generalmente identificarán oportunidades de mejora sobre el sistema, con lo cual se sugerirán cambios a realizar, pero por otra parte si se modifica el sistema es probable que se deban rehacer los *scripts*. ¿Cómo se verifica que el cambio realizado ha dado buenos resultados?

A medida que se adquiere experiencia con las herramientas, los tiempos de automatización van mejorando. Pero, aun así, ésta sigue siendo una de las partes más exigentes de cada proyecto.

Las herramientas especializadas en realizar este tipo de simulaciones ejecutan cientos de procesos, los cuales simulan las acciones que ejecutarían los usuarios reales. Estas herramientas y estos procesos que realizan la simulación se ejecutan desde máquinas dedicadas a la prueba. Las herramientas permiten generalmente utilizar varias máquinas en un esquema *master-slave* para distribuir la carga, ejecutando por ejemplo 200 usuarios desde cada máquina. Incluso existen herramientas que permiten ejecutar la carga desde el Cloud como ya mencionamos.

El principal objetivo de este sistema de distribución de carga es que no podemos dejar que estas máquinas se sobrecarguen, porque de esa forma podrían invalidar la prueba, ya que se generarían problemas para simular la carga o para recolectar los datos de tiempos de respuesta, por ejemplo.

Preparación de la infraestructura de pruebas

De este punto, solo queremos comentar que no hay que desestimar el costo de preparar la base de datos con el volumen definido. Esta es una tarea que generalmente no es fácil de realizar. Hay que pensar en los casos de prueba que se ejecutarán. Por ejemplo, si queremos incluir un total de 2000 ejecuciones de “retiro de dinero” entonces necesitaremos 2000 cuentas con dinero suficiente. Si el sistema bajo pruebas sigue cierto *workflow* y los casos de prueba requieren elementos en la bandeja de entrada de cada usuario a simular, tendremos que generar esas tareas en cada bandeja de entrada de antemano, lo cual generalmente da mucho trabajo. Una opción es usar los *scripts* de pruebas automáticas para generar estos datos. Otra opción es generar los registros directamente en la base de datos. Sea como sea, no se olviden de analizar cuánto tiempo nos llevará esta tarea y quién y cómo la hará.

Por otra parte, considerando que ejecutaremos muchas veces el escenario de prueba, y que una ejecución modifica el estado de los datos de la base de datos,

deberemos recuperar el estado inicial de esta. Es por eso que resulta muy conveniente preparar algún mecanismo automático para recuperar un estado conocido, lo cual generalmente se logra guardando un *back-up* y preparando un *script* para hacer el *restore*.

Por último, deberemos configurar las herramientas de monitorización, y en muchos casos, definir *dashboards* o visualizaciones específicas para poder así facilitar el análisis de los resultados que tenemos y así reducir costos fijos y costos a largo plazo, incluyendo la energía eléctrica que se necesita para tener más servidores o consumir más ciclos de CPU.

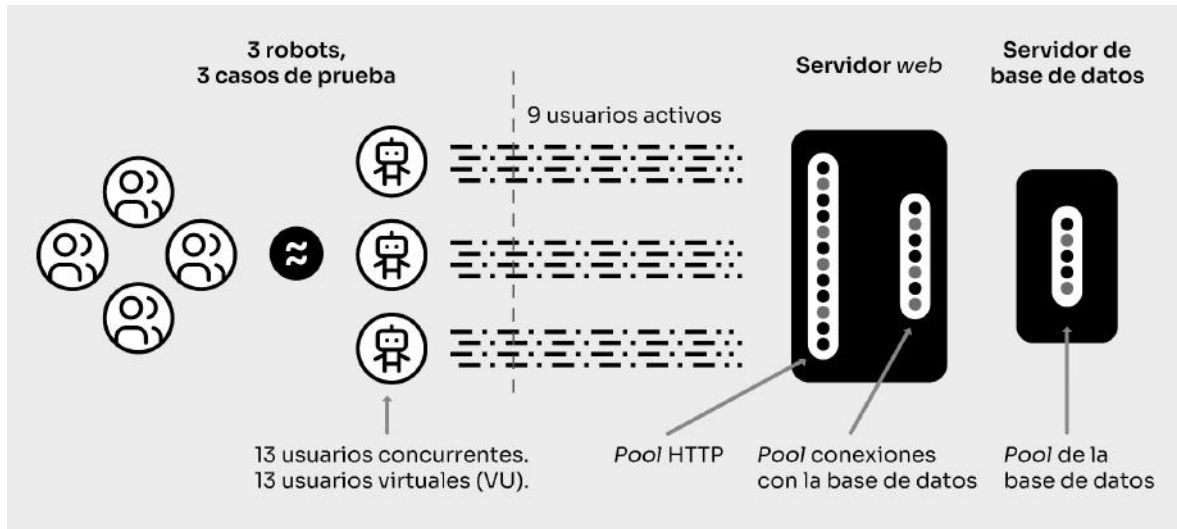
Ejecución de pruebas

Una vez que tenemos todos los artefactos de prueba correctamente instalados, configurados, con todos los casos de prueba automatizados y las herramientas de monitorización listas para recolectar y analizar los datos, comienza la parte más divertida. En esta etapa básicamente ejecutaremos las pruebas simulando así la carga sobre el sistema mientras observamos cómo reacciona. En base al análisis realizado se detectarán oportunidades de mejora, “cuellos de botella”, etc., lo cual permitirá ir “tuneando” la infraestructura y la aplicación hasta alcanzar los niveles de rendimiento definidos como válidos.

Usuario concurrente, usuario activo

Hay algunas aclaraciones a realizar con respecto a los conceptos de *usuarios concurrentes*, *usuarios activos*, *usuarios virtuales*, *usuarios registrados*, etc., que generalmente generan mucha confusión al interactuar con los distintos integrantes del equipo.

Si estamos ejecutando una prueba con 600 usuarios virtuales, en determinado momento podemos tener 500 usuarios actuando, pero solo 300 usuarios activos. En el servidor *web*, tal vez tenemos 200 sesiones activas y en la base de datos tal vez en ese momento se vean solo 15 procesos ejecutando SQLs. ¿Cómo es posible? Muchas veces los administradores de la base de datos están deseando ver muchos procesos ejecutando en su base de datos para ver cómo reacciona esta ante el estrés y, al ver lo que se genera con 600 usuarios, sienten que la prueba no está haciendo lo que debería, que 600 usuarios reales generarían mucha más carga. La siguiente figura consta de una representación que utilizaremos para explicar esta situación.



Usuarios concurrentes, virtuales, activos

Imaginen que un sistema es accedido generalmente por 13 usuarios concurrentes. Estos usuarios siempre ejecutan tres operaciones (casos de uso, funcionalidades o, en este contexto, casos de prueba). Entonces, para simular la carga de estos usuarios se preparan tres *scripts*, o como aparecen en la figura, tres robots que simularán la acción de los usuarios, cada uno ejecutando un caso de prueba distinto. Lo que se hace es poner a ejecutar esos tres *scripts* con 13 instancias, o sea, habrá tantos “usuarios virtuales” como “usuarios concurrentes” o “usuarios reales” en el sistema.

Cada uno de estos usuarios virtuales, con el fin de simular a un usuario real lo más fielmente posible, ejecutará repetidas veces, pero entre repetición y repetición, y entre una acción y otra, se tomará una pausa, las mismas pausas que se toma un usuario para leer la pantalla, llenar un formulario o “pensar”. Por esto, es que a esas pausas se les llama *think time*. El resultado de esto es que en cada momento no habrá 13 usuarios activos, o sea, 13 usuarios realmente ejecutando. En ciertos momentos, algunos estarán ejecutando, otros estarán con pausas y, por eso, vemos que en la figura hay una línea que indica que en ese determinado momento solo hay 9 usuarios activos.

Estas ejecuciones sobre el servidor *web* son atendidas por procesos que leen los pedidos y despachan las ejecuciones sobre el componente adecuado. Este *pool* maneja la concurrencia y es configurable en el servidor *web*. Luego, la lógica de

cada aplicación terminará probablemente comunicándose con la base de datos, lo cual también lo hace con un *pool* de conexiones, tanto del lado del servidor *web* como del lado del servidor de base de datos (también representado en la figura). Como resultado tendremos que, dependiendo del tiempo de respuesta de cada operación ejecutada, tampoco tiene por qué coincidir la cantidad de procesos activos en ese *pool* con la cantidad de usuarios activos. Generalmente, la cantidad de procesos activos del *pool* será menor, pues algunos usuarios estarán realizando otras cosas, como por ejemplo esperando por otros recursos, procesando alguna lógica, etc. Al momento de ejecutar las pruebas será fundamental controlar cómo van evolucionando los usuarios activos y los *pools* de conexiones, principalmente para entender el comportamiento y ayudar al análisis de los “cuellos de botella”. Muchas veces las configuraciones de los distintos *pools* pueden ocasionar grandes problemas.

Bitácora de pruebas

Antes de comenzar a detallar más sobre la ejecución de las pruebas, queremos subrayar algo que consideramos fundamental. Imaginen que venimos ejecutando muchas pruebas y en base a los resultados obtenidos decidimos realizar determinados cambios. Entonces, a medida que avanzamos, se comienzan a probar variantes, se ejecuta una prueba, se cambia otra cosa, se vuelve a ejecutar otra prueba, etc. Además, muchas de las pruebas que ejecutamos no son válidas, porque nos olvidamos de activar algún *log*, reiniciar cierto componente, restaurar la base de datos, etc., entonces las volvemos a ejecutar... y se termina armando un poco de lío.

Quizá un día nos preguntamos: “¿el cambio en la configuración del tamaño en el *pool* de conexiones afectó positivamente a la *performance* de determinada funcionalidad?”. Si no fuimos lo suficientemente ordenados al ejecutar las pruebas y los cambios sobre el sistema, va a resultar muy complicado responder a esta pregunta.

Nosotros seguimos una estrategia muy simple que nos da un muy buen resultado para evitar este tipo de problemas, pero que hay que tener en cuenta desde el

comienzo. Básicamente, la idea es llevar una bitácora para registrar cambios y ejecuciones.

Esto se puede implementar con una planilla de cálculo donde se vayan ingresando ejecuciones y cambios. Las ejecuciones deben referenciar a la carpeta donde se guardaron todos los resultados de monitorización y la carga ejecutada. Los cambios deben ser suficientemente descriptivos como para poder repetirlos y que quede claro qué configuración se estaba utilizando.

De aquí, se obtiene la lista de cambios deseables, los que dieron buenos resultados, los que no, etc.

Ejecución de pruebas iterativas incrementales

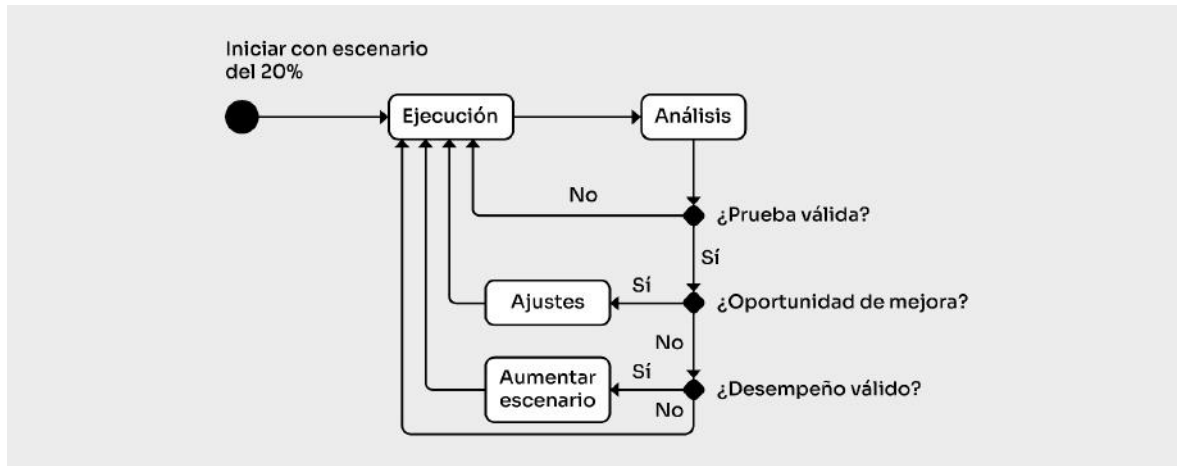
Comenzaremos ahora a hablar propiamente de la ejecución y análisis de resultados, que, como siempre decimos, es la parte más interesante para todo el equipo.

Para comenzar, y para tener un punto de referencia, solemos utilizar una técnica llamada *baseline*. Esta técnica consiste en ejecutar una prueba en la que haya solo un usuario ejecutando un caso de prueba, pues de esa forma se obtendría el mejor tiempo de respuesta que se puede llegar a tener, pues toda la infraestructura está dedicada en exclusiva para ese usuario. Esta prueba debería ejecutarse al menos con 15 datos distintos a modo de tener datos con cierta “validez estadística” y no tener tiempos afectados por la inicialización de estructuras de datos, de cachés, compilación de *servlets*, etc., que nos hagan llegar a conclusiones equivocadas. Además, nos permiten ir analizando los tiempos de respuesta en forma unitaria. Si con un único usuario no conseguimos los tiempos que establecimos al inicio como aceptables, entonces, cuando ejecutemos el escenario de carga, mucho menos. Además, nos servirá para poder comparar el rendimiento del sistema cuando está “bajo carga” y cuando solo tiene un usuario.

Luego pasamos a ejecutar los escenarios de carga. Comenzamos con una pequeña carga, como para verificar también que los *scripts* y el resto de los componentes de prueba estén funcionando bien antes de lanzar la carga objetivo. Si ejecutamos el 100% de la carga al comienzo, puede suceder que observemos muchos problemas al

mismo tiempo, y no sabremos por dónde comenzar. Es más difícil enfrentar esa situación que ir incrementando de a poco el escenario de carga, ya que así los problemas irán “saltando” de a uno. Prestando atención a la siguiente figura, podremos ver cuál es el proceso típico que seguimos en una prueba para ejecutar (luego de haber ejecutado los *baselines*). Generalmente, comenzamos con un 20% de la carga objetivo. El 20% puede ser en base al número de usuarios concurrentes o en base a la frecuencia de ejecución de cada usuario (o sea, si la carga esperada indica que un usuario ejecuta a razón de 10 por minuto, entonces, ejecutaríamos la misma cantidad de usuarios concurrentes, pero a 2 por minuto). Luego de cada ejecución se hace un análisis. Lo primero que hay que determinar es si fue una prueba válida o no, o sea, si se ejecutó correctamente, si la herramienta generadora de carga se comportó correctamente, si se pudieron recolectar todos los datos necesarios sin perder información crucial para el análisis de desempeño de la infraestructura, etc. De no ser así, habría que repetir la prueba. Si la prueba se puede considerar válida, entonces, el análisis se seguirá en búsqueda de oportunidades de mejora, esto es, determinar “cuellos de botella”, ver qué componentes se podrían optimizar, si algún indicador está por fuera de lo aceptable, etc. En caso de que alguna oportunidad de mejora se detecte, debería implementarse y, para verificar que realmente se logre la mejora esperada, habría que ejecutar la misma prueba para poder comparar. Muchas veces se piensa que un cambio en determinados parámetros puede mejorar el rendimiento, y no siempre es así, por esto es necesario verificarlo empíricamente antes de continuar. Una vez que estos cambios se implementaron y se verificó su mejora, entonces pasaríamos a incrementar el escenario de carga.

Este ciclo se continúa hasta llegar al 100% de la carga esperada. Si se cuenta con tiempo (y voluntad), se podría continuar aumentando la carga como para asegurarnos más aún. No es algo necesario ni obligatorio, pero muchas veces, ya que tenemos todo preparado, podemos obtener más información. Hay oportunidades en las que una prueba de carga pasa a ser una prueba de estrés, o sea, se sigue incrementando la carga hasta encontrar el punto de saturación máxima, conociendo así el límite de usuarios que soporta la infraestructura actual. O podríamos ejecutar otros tipos de pruebas, como pruebas de picos o *endurance*.



Workflow para guiar la ejecución

Agilizando las pruebas de *performance*

Es fundamental tener como objetivo la entrega temprana de resultados para disminuir el costo de las pruebas y permitir que se implementen las mejoras lo antes posible.

El enfoque tradicional de pruebas de *performance* (que se puede visualizar en la siguiente figura) plantea un proceso en cascada, en donde no se comienza a ejecutar pruebas hasta que se hayan superado las etapas previas de diseño e implementación de las pruebas automatizadas. Una vez que está listo, el proceso de ejecución se inicia y, recién a partir de ese momento, el cliente comienza a recibir resultados. Es decir: desde que el cliente (o *stakeholder*) indicó la necesidad y deseo de realizar las pruebas, hasta que recibe un primer resultado para analizarlo e implementar mejoras (en algoritmos, *tuning* de la infraestructura, etc.) pueden pasar dos semanas. Por más que parezca poco, esto es demasiado para las dimensiones de un proyecto de *performance*, que generalmente no dura más de uno o dos meses.

Es necesario agilizar este proceso tradicional para brindar resultados lo antes posible. Si logramos esto, el equipo de desarrollo podrá realizar mejoras cuanto antes, tendremos una mejor visión del cliente y estaremos verificando la validez de

nuestras pruebas, dando el tipo de resultados que espera obtener nuestro cliente. Si tenemos algún error en el análisis o diseño de las pruebas y nos enteramos de esto al dar los resultados, será muy costoso de corregir, de la misma forma que un *bug* es más costoso para un equipo de desarrollo cuando se detecta en producción.

Veamos entonces algunas ideas de cómo agilizar este proceso.

Entregar resultados desde que estamos automatizando

Si bien el foco de la automatización no es el de encontrar errores o posibles mejoras al sistema, si estamos atentos y prestamos atención, podemos adelantar varias posibilidades de mejora.

Esto es posible dado que la automatización implica trabajar a nivel de protocolo y se termina conociendo y analizando bastante el tráfico de comunicación entre el cliente y el servidor. Por ejemplo, en el caso de un sistema *web*, hemos llegado a encontrar posibilidades de mejora tales como:

- Manejo de *cookies*.
- Problemas con variables enviadas en parámetros.
- Problemas de seguridad.
- HTML innecesario.
- Recursos inexistentes (imágenes, CSS, JavaScript, etc.).
- *Redirects* innecesarios.
- ¡E, incluso, *bugs* funcionales!

También es cierto que el equipo de desarrollo pudo haber realizado ciertas pruebas para no llegar con estos errores a la etapa de desarrollo; sin embargo, la experiencia muestra que en la mayoría de los proyectos se encuentran muchos de estos incidentes. Por eso, al realizar estas pruebas estamos entregando resultados de valor que mejorarán la *performance* del sistema.

Ejecuciones tempranas

Para poder comenzar a ejecutar algo no es necesario esperar y contar con toda la prueba armada. Nosotros solemos ejecutar pruebas que incluyen un solo caso de prueba (o incluso, una porción, como por ejemplo solo el acceso a la URL base y al

login) con muchos usuarios, a modo de analizar cómo se comporta cada caso de prueba en específico y sin mezclarlo con el resto, podemos focalizar el análisis. Estas pruebas son de gran valor porque permiten encontrar problemas de concurrencia, bloqueos de acceso a tablas, falta de índices, utilización de caché, tráfico, etc.

Además, esto nos da una medida de comparación: podemos ver cuál es el tiempo de respuesta que tiene el caso de prueba funcionando “solo”, para luego comparar estos tiempos con los de la prueba en concurrencia con otros casos y ver cómo impactan unos sobre otros (tal como ya habíamos comentado sobre los *baselines*). Lo que suele pasar es que algunos casos de prueba bloqueen tablas que otros utilizan y, de esa forma, repercutan en los tiempos de respuesta, al estar en concurrencia. Esta comparación no la podríamos hacer tan fácil si ejecutamos directamente la carga completa.

Cuando lleguemos a completar la automatización de todos los casos de prueba, el equipo de desarrollo ya puede estar trabajando en las primeras mejoras que les fuimos reportando, habiendo así solucionado los primeros problemas que hubiesen aparecido al ejecutar pruebas. Otra ventaja es que las pruebas con un solo caso de prueba son más fáciles de ejecutar que las pruebas con 15 casos de prueba, pues es más fácil preparar los datos, analizar los resultados e incluso analizar cómo mejorar el sistema, ya que cuando tenemos 15 funcionalidades es más complejo el análisis, hay más datos para filtrar y buscar la información que nos de la pista de por dónde atacar el problema.

¿Cómo disminuir los problemas detectados en una prueba de *performance*?

Es típico que las pruebas de *performance* se dejen para el final de un proyecto (si hay tiempo y presupuesto), sin haber investigado nada en absoluto con anterioridad. Esto hace que se encuentren todos los problemas relacionados a la *performance* al final, juntos.

Pensemos qué pasa con respecto a las pruebas de aspectos funcionales. En general (lamentablemente, no siempre) se hacen pruebas unitarias por parte de cada desarrollador antes de integrar el sistema. Una vez integrado se hacen pruebas de

integración y ahí se le pasa una nueva versión al equipo de pruebas para que hagan las pruebas de sistema.

Creemos que este enfoque aplicado a la *performance* (y, obviamente, a muchos otros aspectos no-funcionales también) podría hacer que la aplicación llegue mucho más depurada al momento de la prueba final en la plataforma definitiva.

Generalmente, los desarrolladores no consideran la *performance* desde el comienzo de un proyecto; con un poco de esfuerzo extra se podrían solucionar problemas en forma temprana, y hablamos de problemas graves: aspectos de la arquitectura del sistema, mecanismos de conexión, estructuras de datos, algoritmos.

No parece muy difícil implementar un programa que ejecute muchas veces un método y que registre los tiempos. Pero, sin llegar a implementar algo tan específico, podríamos al menos trabajar las pruebas de *performance* a nivel de API, lo cual permite realizar experimentos mucho más temprano que al tener el sistema completo en todas sus capas.

Esta práctica nos puede anticipar una enorme cantidad de problemas, que serán mucho más difíciles de resolver si los dejamos para una última instancia. Si sabemos que vamos a ejecutar pruebas de *performance* al final del desarrollo, tenemos una tranquilidad muy grande. Pero tampoco es bueno confiarse y esperar hasta ese momento para trabajar en el rendimiento del sistema. Será posible mejorar el desempeño de todo el equipo y ahorrar costos del proyecto si se tiene en cuenta estos aspectos en forma temprana, desde el desarrollo.

Algunas ventajas de este enfoque:

- Mayor compromiso de los desarrolladores con la *performance*; verifican y no liberan módulos con problemas de *performance* graves.
- Familiarización de los desarrolladores con el uso de recursos de sus sistemas; descubren qué cosas de su código generan problemas y seguramente les ayude a evitarlo a futuro.
- Menor riesgo de tener problemas de *performance*.

- Detección temprana de los problemas, con lo cual será más barato resolverlos.

Y, a su vez, podríamos listar algunas desventajas:

- Lapso mayor para liberar la primera versión.
- Necesidad de aprender a utilizar herramientas de monitorización.

Claro que esto no es para hacerlo con cada método, API o componente que se desarrolle, pero se podría pensar en cuáles serán los más utilizados, los que tienen las tareas más demandantes en cuanto a consumo de recursos, los que deben procesar más datos, etc.

También debemos dejar claro que esto no sustituye las pruebas de *performance*, sino que nos prepara mejor para ellas. No las sustituye porque no se prueban las distintas funcionalidades en conjunto, no se prueba la operativa, no se prueba en un servidor similar al de producción, sino que en el de desarrollo, etc.

Pruebas de *performance* en integración continua

Este tipo de enfoques ha tomado mayor relevancia y facilidad de adopción al considerar que se ha adoptado muchísimo la integración continua de la mano de las metodologías ágiles⁵⁵.

La forma de incorporar verificaciones de *performance* en este enfoque es ejecutando ciertos test en lo que se conoce como el *pipeline* en nuestra herramienta de integración continua (básicamente, se trata de una secuencia de acciones que se ejecutan con tal de construir el producto a partir de los últimos cambios del código, ejecutando también una serie de verificaciones para detectar lo antes posible si estos cambios introdujeron algún problema). Esto se puede hacer cada vez que se hace el *build* o, si vemos que toma mucho tiempo, fijar ciertos horarios frecuentes.

⁵⁵ Para profundizar sobre integración continua: <<https://www.federico-toledo.com/diferencia-entre-continuous-integration-delivery-y-deployment>>

Las particularidades de este enfoque, entonces, son:

- Se ejecuta frecuentemente. Se busca identificar si los últimos cambios afectan negativamente la *performance* (comparando contra las ejecuciones previas), teniendo *feedback* temprano. No se busca validar si la carga esperada va a ser soportada por la infraestructura actual.
- No se simula producción, sino que se intenta ejecutar pruebas tempranas, más “baratas” y por sobre todo, fáciles de mantener.
- Se hacen, por esto último, a nivel de servicios (a nivel de la API).
- No se ejecuta en la infraestructura similar a producción, sino que se busca hacer en una infraestructura reducida.

Pueden leer más al respecto aquí⁵⁶.

Pruebas de *performance* en producción

Imaginen que tenemos que ejecutar una prueba la cual ya diseñamos, con un buen escenario de carga, casos de prueba, datos de prueba, todo automatizado y preparado en la herramienta de simulación de carga que vamos a usar. **Pero** no tenemos una infraestructura de pruebas que se compare a la de producción. Esto suele suceder cuando la infraestructura de producción incluye un servidor de gran porte, como puede ser un AS400 o cualquier otro tipo de *mainframe*, un clúster compuesto por muchas máquinas o con una configuración compleja y cara como para preparar un ambiente exclusivo para *testing*.

¿Dónde ejecutamos la prueba? Seguramente, el servidor de desarrollo o el destinado para pruebas funcionales no cuentan con las características que tiene el *server* de producción. ¿Qué alternativa tenemos ante este problema? Pues, ¡probar en producción!

Muchas veces nos han preguntado si se puede probar en producción. ¿Cuál es el problema? Simple: debemos asumir muchos riesgos. ¿Qué pasa si al ejecutar una prueba de este estilo sobre el sistema bajo pruebas afectamos al resto de los

⁵⁶ Pruebas de *performance* en integración continua

<<https://www.federico-toledo.com/pruebas-de-performance-en-integracion-continua>>

sistemas que están en producción? Más de una vez nos han dicho que los AS400 no se caen, pero esto no es así y lo hemos vivido en más de una situación. Sí, en producción. Ese es el principal riesgo.

Por otra parte, un problema importante es que los resultados que obtenemos están impactados por el resto de las aplicaciones que están en producción en el mismo servidor.

De todos modos, esta aproximación tiene una gran ventaja: nos aseguramos de que la infraestructura de prueba es similar a la de producción, en realidad, es exactamente la misma. En esto estamos incluyendo todo lo que refiere a *hardware*, *software* de base (incluidas las versiones de cada componente, sus configuraciones particulares, etc.) y el resto de los sistemas con los que tendría que convivir en producción (desde antivirus hasta otros sistemas de información de la empresa).

Entonces, lo que creíamos que era un grave problema, en realidad, es la situación ideal.

La mejor estrategia para reducir esos riesgos sería hacer pruebas iniciales en otra máquina, desarrollo, *testing* o donde se pueda. Luego, intentar resolver los problemas que ahí surjan bajo la carga que se pueda llegar a simular (que, seguramente, será mucho menor que la que queremos simular), y luego pasar a producción. Quien controla la prueba debe tener bajo la vista los indicadores básicos: CPU, memoria, disco y red, tener claro qué umbrales dejan de ser aceptables y, una vez que se llegue a esos niveles, detener la prueba.

También podemos considerar ejecutar pruebas en horarios en los que hay menor carga, incluso de noche o los fines de semana. El problema de esto es que no es solo una persona que se necesita para presionar el botón *play* a la herramienta de simulación y nada más. Si realmente queremos obtener buenos resultados de una prueba de *performance*, es necesario tener a expertos de infraestructura, algún representante del equipo de desarrollo, etc.

Otro aspecto a considerar si vamos a ejecutar en producción son los datos de prueba. Si no podemos tener una base de datos aparte, una estrategia que se usa es tener datos de prueba en la misma base de producción pero que estén “lógicamente” divididos (esto lleva un diseño para que no afecte los datos reales, pero puede ser, quizá, más fácil de lograr en ciertos contextos). La otra opción es

solo probar consultas, y no probar casos de prueba que impliquen escritura o modificación de datos. Eso tiene su contra, que es que nos quedan cosas sin probar y son las que más hacen mover al disco, las que generalmente son bloqueantes a nivel de estructuras de datos, etc., o sea, quedará aún mucha incertidumbre, mucho riesgo, pero es mejor que nada.

Pruebas de componentes mediante *benchmarks*

Un punto más que queremos mencionar relacionado a las pruebas de *performance* son las pruebas de los componentes de la infraestructura mediante el enfoque que se conoce como *benchmark*. Un *benchmark*, o prueba de referencia, es un estándar o medida de desempeño que se utiliza para comparar y evaluar el rendimiento relativo de un sistema. Los *benchmarks* son útiles en pruebas de *performance* para establecer una línea base y determinar la eficiencia y calidad de un sistema en comparación con otros similares en condiciones específicas.

La recomendación es poder ejecutar los *benchmarks* de ciertos componentes de la infraestructura de manera anticipada a nuestras pruebas, para poder detectar posibles desvíos o problemas de antemano, lo cual ayuda a eliminar una posible causa de “ruido” en las pruebas de *performance* y ejecutar las pruebas de *performance* con los componentes en el estado correcto.

Ejecutar un *benchmark* es relativamente barato comparado a una prueba de *performance* típica. Consta en ejecutar una serie de programas que brindan información y comparativas. Las herramientas para la ejecución y diagnóstico de un *benchmark* las proporcionan los propios fabricantes o es posible conseguir alguna implementada por parte de terceros.

Lo más complejo de realizar *benchmark* es comparar los valores obtenidos contra los valores esperados, esto es esencial para poder diagnosticar si son valores correctos o no. La validez se compara contra la especificación del componente, en base a nuestra experiencia previa, comparaciones contra otros test realizadas por terceros o incluso frente a los componentes de otros fabricantes.

Los fabricantes de componentes generalmente venden diciendo que lo que ofrecen es lo mejor que hay, y que no hay posibilidades de tener problemas. Luego, tienen un sistema de soporte que ayuda a resolver los problemas una vez que existen y para eso ¿vamos a esperar a tenerlos en producción? Es mucho mejor detectarlos de antemano.

Es una tarea principalmente ejecutada por un analista de infraestructura, pero es también una tarea de *testing*. Algunos de los componentes generalmente puestos a prueba son (y observar que son tanto componentes de *hardware* como de *software* de base):

- CPU/FPU/GPU
- Memoria
- Disco
- Placa de red
- Base de datos
- *Server* (*WebServer*, *MailServer*, *FileServer*)
- Kernel del Sistema
- Proxy/DNS

Cuando un problema es detectado, dependiendo de qué tipo de problema sea, es resuelto por cambios en configuración, actualización de *firmware/drivers/hotfix/updates* o se termina elevando el problema al soporte del componente, al servicio oficial o mismo al fabricante del componente.

Al ser componentes en donde su fabricación participan terceros, existe una mayor dependencia para su resolución y es por eso que se recomienda ejecutar este tipo de test lo antes posible, antes de detectarlos en las pruebas de *performance* o en producción, ya que pueden existir retrasos hasta lograr solventar los posibles problemas.

Falacias del *Testing* de *Performance*

Es interesante ver que hay muchas formas en las que uno puede equivocarse, pero las que aquí queremos destacar son aquellas que ya son falacias, en el sentido de que son creencias que nos llevan a actuar en forma equivocada, pero creyendo que estamos haciendo lo correcto.

Jerry Weinberg (2008), en su libro, *Perfect software and other illusions about testing*, explica muchas falacias relacionadas al *testing* en general. Nosotros queremos aportar con algunas específicas de las pruebas de *performance*.

Falacias de planificación

Muchas veces se piensa que las pruebas de *performance* tienen lugar solo al final de un proyecto de desarrollo, justo antes de la puesta en producción, como para hacer los últimos ajustes (*tuning*) y hacer que todo vaya rápido. Así es que se ve al *Testing* de *Performance* como la solución a los problemas de *performance*, cuando en realidad es la forma de detectarlos, anticiparse a ellos y comenzar a trabajar para resolverlos. El mayor problema es que, si recién al final del proyecto tenemos en cuenta la *performance*, podemos encontrar problemas muy grandes que serán muy costosos de reparar. Lo ideal sería, como ya lo hemos indicado, considerar la *performance* desde las etapas tempranas del desarrollo, realizando ciertas pruebas intermedias que nos permitan anticiparnos a los problemas más importantes que puedan surgir.

Falacia de “más fierro”

Cuántas veces habremos oído que las pruebas de *performance* no son necesarias, pues si se detectan problemas de rendimiento en producción lo podemos resolver

simplemente agregando “más fierros”, o sea, incrementando el *hardware*, ya sea agregando servidores, agregando memoria, etc. Piensen en el caso de un *memory leak*, si se agrega más memoria, tal vez logramos que el servidor esté activo durante 5 horas en lugar de 3, pero, el problema, no lo resolvemos así. Tampoco tiene sentido aumentar los costos en infraestructura si se puede ser más eficiente con lo que tenemos y así reducir costos fijos y costos a largo plazo, incluyendo la energía eléctrica que se necesita para tener más servidores y consumiendo más ciclos de CPU.

Falacias de entorno de pruebas

Relacionado también a “los fierros”, tenemos esta otra falacia, en la que se afirma que podemos realizar las pruebas en el entorno de *testing*, sin importar qué tanto se parece este al entorno de producción. O quizá, para un cliente hicimos las pruebas en Windows y creemos que la aplicación funcionará perfectamente en otro cliente que instalará el sistema bajo Linux. Al hacer las pruebas, tenemos que ser cuidadosos de hacerlo sobre un entorno que sea lo más parecido al de producción. Hay muchos elementos del entorno que influyen en la *performance* de un sistema, desde los componentes de *hardware*, las configuraciones del sistema operativo, cualquiera de los componentes con los que interactúe e incluso el resto de las aplicaciones que estén ejecutando al mismo tiempo.

De igual forma, podemos pensar sobre la base de datos. Hay quienes creen que se puede realizar una prueba de *performance* con una base de datos de pruebas. Si tenemos problemas con las consultas SQL, quizá pasen desapercibidos, pero si luego en producción tenemos una base de datos con miles de registros, seguramente, los tiempos de respuesta de SQLs que no están optimizadas nos den grandes problemas.

Falacias de predicción de rendimiento por comparación

Peor que la falacia del entorno de pruebas es hacer las pruebas en un entorno y sacar conclusiones para otros entornos. No deberíamos extrapolar resultados. Por ejemplo: “Si duplico servidores, entonces, duplico velocidad”; “si aumento memoria, entonces, aumenta cantidad de usuarios que soporta”, etc. Este tipo de afirmaciones no se puede hacer. Generalmente, hay muchos elementos que impactan en el rendimiento global. La cadena se rompe por el eslabón más frágil. Si mejoramos 2 o 3 eslabones, el resto seguirá siendo igual de frágil. Dicho de otra forma, si aumentamos algunos de los elementos que limitan la *performance* de un sistema, seguramente, el “cuello de botella” pase a ser otro de los elementos que están en la cadena. La única forma de asegurarse es probando.

Tampoco es válido hacer una extrapolación en el otro sentido. Imaginen que si tenemos un cliente que tiene 1000 usuarios ejecutando con un AS400 y le funciona perfectamente, no podemos pensar en cuál es el *hardware* mínimo necesario para darle soporte a 10 usuarios. Es necesario probarlo para verificarlo.

Falacia del *Testing* Exhaustivo

Pensar que con una prueba de *performance* se evitan todos los problemas es un problema. Cuando hacemos las pruebas buscamos (por restricciones de tiempo y recursos) detectar los problemas más riesgosos y con mayor impacto negativo. Para lograr esto, se suele acotar la cantidad de casos de prueba (generalmente, como ya mencionamos, no más de 15). Es **muy** costoso hacer una prueba de *performance* donde se incluyan todas las funcionalidades, todos los flujos alternativos, todos los juegos de datos, etc. Esto implica que siempre podrán existir situaciones no probadas que produzcan, por ejemplo, algún bloqueo en la base de datos, tiempos de respuesta mayores a los aceptables, etc. Lo importante es cubrir los casos más comunes, los más riesgosos, etc. y que cada vez que se detecte un problema se intente aplicar esa solución en cada punto del sistema donde pueda impactar. Por ejemplo, si se encuentra que se están manejando inadecuadamente las conexiones

de la base de datos en las funcionalidades que se están probando, una vez que se encuentre una solución, se debería aplicar a todo el sistema, en todos los lugares donde se manejan conexiones. Muchas veces una solución es global, como la configuración del tamaño de un *pool* o de la memoria asignada en la *Java Virtual Machine (JVM)*.

Otra cosa muy recomendable para dar mayor tranquilidad con respecto a la *performance* implica monitorizar el sistema en producción para detectar lo antes posible los problemas que puedan surgir por haber quedado fuera del alcance de las pruebas y así corregirlos lo antes posible, antes de que sean detectados por los usuarios.

Falacias tecnológicas

¿Cuál es una de las principales ventajas de los lenguajes modernos como Java o C#? No es necesario hacer un manejo explícito de memoria gracias a la estrategia de recolección de basura (*garbage collection*) que implementan los *frameworks* sobre los que ejecutan las aplicaciones. Entonces, *Java o C# me garantizan que no habrán memory leaks*. Lamento informar que esto es **falso**. Hay dos situaciones, principalmente, donde podemos producir *memory leaks*, una es manteniendo referencias a estructuras que ya no usamos, por ejemplo, si tenemos una lista de elementos, y siempre agregamos pero nunca quitamos y, por otro lado, el caso de que la JVM o *framework* de Microsoft tengan un *bug*, por ejemplo, en alguna operación de concatenación de *strings* (este caso nos pasó en un proyecto concreto). Por esto, es tan importante prestar atención al manejo de memoria y utilizar herramientas que nos permitan detectar estas situaciones, tales como los *profilers* para Java⁵⁷.

Muy relacionado a esto, existe otra falacia con respecto al manejo de memoria en estas plataformas, que es pensar que al forzar la ejecución del *Garbage Collector* (GC) invocando a la funcionalidad, explícitamente, se libera la memoria antes y de esa forma se mejora la *performance* del sistema. Esto no es así, principalmente

⁵⁷ *Profiler* para Java <<http://java-source.net/open-source/profilers>>

porque el Full GC (cuando se invoca explícitamente) es un proceso que bloquea la ejecución de cualquier código que se esté ejecutando en el sistema, para poder limpiar la memoria que no está en uso. Realizar esta tarea lleva tiempo. Las plataformas como la JVM y el *framework* de Microsoft tienen algoritmos donde está optimizada la limpieza de la memoria, no se hacen en cualquier momento, sino que en situaciones especiales que tras experimentar han llegado a la conclusión de que es lo más eficiente a lo que se puede llegar. Hay formas de ajustar el comportamiento de estos algoritmos, ya que de acuerdo al tipo de aplicación y al tipo de uso está comprobado que distintas configuraciones dan distintos resultados, pero esto se logra a base de ajustar parámetros de configuración y no invocando explícitamente por código la ejecución del GC.

También hemos visto que es común pensar que el caché es un método rápido y fácil de optimizar una aplicación, que simplemente con configurar ciertas consultas en el caché ya vamos a mejorar nuestra aplicación, y listo, sin siquiera evaluar antes otras opciones. El caché es algo muy delicado y, si no se tiene cuidado, puede incorporar más puntos de falla. Cuando se pierde el caché, si no está optimizada la operación, puede provocar inestabilidad. Es necesario hacer una verificación funcional de la aplicación, viendo si al configurar el caché en determinada forma no estamos cambiando el comportamiento esperado del sistema, ya que las consultas no nos darán los datos completamente actualizados. Es necesario medir el *cache hit/miss* así como el costo de refresco y actualización y así analizar qué consultas nos darán más beneficios que complicaciones.

Falacias de diseño de pruebas

Si probamos las partes, entonces estaremos probando el total. Esta es una falacia mencionada por Jerry Weinberg para el *testing* en general, pero aquí se hace mucho más visible que esa afirmación no es correcta. No podemos hacer una simulación sin tener en cuenta los procesos o la operativa en general, enfocándonos en probar unitariamente y no concurrentemente diferentes actividades. Quizá probamos una operación de “extracción de dinero” con 1000 usuarios, por otro lado, una de “depósito de dinero” con 1000 usuarios, pero no estamos garantizando nada de que las dos operaciones al trabajar en concurrencia no tendrán problema.

Si entre ellas tienen algún problema de bloqueos, entonces, quizá con un total de 10 usuarios ya presenten graves problemas en los tiempos de respuesta.

Hay dos falacias casi opuestas aquí y lo que nosotros consideramos “correcto” o “más adecuado” es encontrar el punto medio. Hay quienes piensan que probando cientos de usuarios haciendo “algo”, quizá todos haciendo lo mismo, ya estamos haciendo una prueba correcta. Y, por otro lado, hay quienes creen que es necesario incluir todo tipo de funcionalidad que se puede ejecutar en el sistema. Ninguna de las dos es válida. Una porque es demasiado simplista y deja fuera muchas situaciones que pueden causar problemas y la otra por el costo asociado, es muy caro hacer una “prueba perfecta”. Tenemos que apuntar a hacer la mejor prueba posible en el tiempo y con los recursos que contamos, de forma tal de evitar todos los contratiempos con los que nos podemos encontrar. Esto también incluye (si hay tiempo y recursos) simular casos que pueden ocurrir en la vida real, tales como borrar cachés, desconectar algún servidor, generar “ruido” de comunicación, sobrecargar servidores con otras actividades, etc., pero claro está que no podemos probar todas las situaciones posibles, ni tampoco hacer la vista a un lado.

Falacia del vecino

Se suele caer en el pensamiento de que las aplicaciones que ya las está usando otro sin problemas no me ocasionarán ningún problema cuando yo las use. Entonces, no tengo que hacer pruebas de *performance* porque mi vecino ya tiene el mismo producto con un uso mayor y le “camina” todo bien. Como se dijo antes, no se puede extrapolar ningún resultado de un lado para otro. Por más que el hecho de que el sistema esté en funcionamiento con cierta carga de usuarios, es necesario quizá afinarlo, ajustar la plataforma, asegurar que los distintos componentes están correctamente configurados y que la *performance* será buena bajo el uso que le darán nuestros usuarios a ese sistema.

Falacia de exceso de confianza

Esto sucede en general, pero también con respecto a la *performance* de los sistemas. Se tiende a pensar que los sistemas que tendrán problemas serán solo esos desarrollados por programadores que se equivocan, que no tienen experiencia, etc. “Como mis ingenieros tienen, todos, muchos años de experiencia, no necesito probar la *performance*, porque no es la primera vez que desarrollan un sistema de gran porte. Por supuesto que va a funcionar bien”. O quizá la afirmación viene por el lado de “nunca tuvimos problemas, ¿por qué vamos a tener ahora?”. No hay que olvidar que programar es una tarea compleja y, por más experiencia que se tenga, errar es humano, más al desarrollar sistemas que estén expuestos a múltiples usuarios concurrentes (que es lo más común) y su rendimiento se vea afectado por tantas variables: tenemos que considerar el entorno, plataforma, máquina virtual, recursos compartidos, fallos de *hardware*, etc., etc.

Otro problema en el que se cae al tener un exceso de confianza, se da cuando se están haciendo pruebas de *performance*. Generalmente, la ejecución de pruebas se sugiere hacerla en forma incremental. O sea, comenzamos ejecutando un 20% de la carga total que queremos simular, para poder así “atacar” los problemas más graves primero, y luego ir escalando la carga a medida se van ajustando los incidentes que se van encontrando. Pero hay quienes prefieren arrancar directamente con el 100% de la carga para encontrar problemas más rápido. La dificultad de ese enfoque es que todos los problemas salen al mismo tiempo y es más difícil poner foco y ser eficiente en repararlos.

Falacias de la automatización

Ya que estamos mencionando falacias relacionadas a las pruebas en sí, he aquí otra muy común que genera grandes costos. Se tiende a pensar que los cambios en la aplicación bajo pruebas que no se notan a nivel de pantalla no afectan la automatización, o sea, que no afectan a los *scripts* que tenemos preparados para simular la carga del sistema. Generalmente, al realizar cambios en el sistema, por más que no sea a nivel de interfaz gráfica, hay que verificar que los *scripts* de

pruebas que tenemos preparados sigan simulando la ejecución de un usuario real en forma correcta, pues, de no ser así, podemos llegar a sacar conclusiones erróneas. Si cambian parámetros, la forma en que se procesan ciertos datos, el orden de invocación de métodos, etc., puede hacer que el comportamiento simulado deje de ser fiel a la acción de un usuario sobre el sistema bajo pruebas o directamente hacer que los *scripts* fallen.

Comentarios finales del capítulo

Sin duda, las pruebas de *performance* son muy importantes y nos dan una tranquilidad muy grande a la hora de poner un sistema en producción. Mientras más temprano obtengamos resultados, más eficientes seremos en nuestro objetivo de garantizar el rendimiento (*performance*) de un sistema.

Para aportar valor en una prueba de *performance* se necesita sumar mucha experiencia en pruebas y tener una vocación muy grande por entender “a bajo nivel” cómo funcionan las cosas. Esta variedad de *skills* que se necesitan para este tipo de rol es una de las cosas que lo hace apasionante. Siempre aparecen tecnologías nuevas, siempre hay soluciones más complejas, ¡siempre hay un desafío! Nosotros muchas veces decimos que en este tipo de proyectos jugamos al *Doctor House*⁵⁸, ya que una tarea fundamental es poder diagnosticar precisamente qué le pasa al sistema y sugerir posibles soluciones. Y jugando a los médicos, debemos distinguir **síntomas** de **causas**. Nosotros queremos observar los síntomas, en base a nuestra experiencia, ser capaces de determinar las causas y así poder recetar una solución a nuestro paciente, a nuestro sistema bajo pruebas. Muchas veces, esto se hace preguntando reiteradas veces “¿por qué sucede este síntoma? Por ejemplo:

- Se observa que el tiempo de respuesta de una funcionalidad es mayor al aceptable.
 - ¿Causa o síntoma?
 - Síntoma. Entonces, ¿por qué sucede?
- Se analizan los tiempos discriminando cada acción de esa funcionalidad y se observa que el tiempo de respuesta de un componente de la aplicación se lleva la mayor parte.
 - ¿Causa o síntoma?
 - Síntoma. Entonces, ¿por qué sucede?

⁵⁸ Doctor House (2004-2012), famosa serie televisiva: <<http://www.imdb.com/title/tt0412142>>

- Se analiza ese componente y se observa que el tiempo de respuesta de una SQL ejecutada en ese componente es muy lento para lo que hace.
 - ¿Causa o síntoma?
 - Síntoma. Entonces, ¿por qué sucede?
- Luego de un análisis en conjunto con el encargado de la base de datos se observa la falta de un índice en la base de datos.
 - ¿Causa o síntoma?
 - Causa. Entonces, ¿cómo se soluciona?

Pasar de una causa a una solución generalmente es fácil, al menos para una persona con experiencia en esa área. En este caso, el experto de base de datos agregaría un índice adecuado y esto resolvería la causa, que haría que se evitaran los síntomas presentados. De esta forma, con esta prueba, solucionamos el problema que detectamos. De otro modo, los usuarios hubiesen sufrido esas demoras, el experto de base de datos hubiese recibido muchas quejas y tendría que haber detectado el problema y ajustado estos parámetros en producción con los riesgos e impactos que esto pueda significar.

En otras palabras, no dejen para mañana las pruebas que puedan hacer hoy.

“...no hay tres minutos, ni hay cien palabras que me puedan definir.”

Roberto Musso

HABILIDADES PARA *TESTING*

Porque no todo son técnicas y herramientas para el éxito de un equipo de pruebas, queremos también abordar algunos temas más relacionados con los aspectos humanos a ser tenidos en cuenta. Hay quienes hacen referencia a estos temas bajo el paraguas de *soft skills* o *habilidades blandas*. Si bien es algo muy popular, hace años se cuestiona el término, ya que da lugar a concebir que son menos importantes. Por esto, algunas formas alternativas de llamarle pueden ser *core skills* o *habilidades transversales, centrales o profesionales*.

¿Qué habilidades necesitamos desarrollar para *testing*?

A lo largo de estos capítulos les hemos presentado conceptos básicos de *testing*, cómo diseñar casos de prueba, pruebas automatizadas y de *performance*, etc., que con seguridad servirán como preparación y también como guía para quien haga pruebas. Ahora bien, basándonos en nuestra experiencia, de la mano de estas técnicas es necesario desarrollar otras habilidades que hacen a un buen *tester*. Intentaremos dar un acercamiento a estas habilidades personales (*transversales*, muchas veces mal llamadas *habilidades blandas*) que son también importantes y que hay que reforzar a medida que crecemos como profesionales.

Habilidad de comunicación

Comunicamos desde las palabras que utilizamos y desde el tono de voz; comunicamos desde el contacto visual, el saludo, la postura e incluso desde los movimientos que hacemos con las manos o brazos. Todo es un conjunto que tiene como fin establecer contacto con el otro para transmitir cierta información y, según sea nuestra habilidad, podemos facilitar o dificultar el proceso. Nuestra forma de comunicar hace a nuestra imagen, nos aleja o nos acerca a los *stakeholders* y es algo que puede cuidarse prestando atención. Tan simple como prestar atención a cosas sencillas que podremos ejercitar: escuchar, parafrasear, hablar claro, hacer buenas preguntas, etc.

Escuchar requiere un gran esfuerzo. Es común estar pendiente de nuestros pensamientos, en un diálogo con nosotros mismos, a ver qué vamos a decir: “Y ahora la otra persona dice esto y yo puedo contestar lo otro...”, es necesario aprender a escuchar activamente. ¿Qué significa *escuchar activamente*? Se trata de escuchar intentando comprender profundamente lo que nos quieren transmitir. Es importante no distraerse, no juzgar, no subestimar, no interrumpir, aún si nos

estuvieran contando un problema y a la mitad del mensaje se nos ocurriera una solución, podría resultar mejor dejar hablar al otro en lugar de estar impacientes.

Otro punto importante al escuchar es mostrar empatía, o sea, tratar de “ponernos en los zapatos del otro”. No se trata de ser simpático, tampoco implica estar de acuerdo con la otra persona, sino que se trata simplemente de mostrar que lo entendemos. Por supuesto que la comunicación no verbal debe ir acompañando la verbal: mantener el contacto visual, no cruzarse de brazos, no mirar por la ventana, si no, no lograremos generar esa empatía que buscamos.

Parafrasear es decir con nuestras palabras lo que acabamos de escuchar. Esto permite hacerle saber al otro que lo escuché, verificando si ambos hemos entendido lo mismo. Es un ejercicio muy útil cuando hay muchos actores construyendo un *software*, con lo cual evitamos malentendidos. Luego de resumir a la otra persona lo que hemos entendido, podremos pedir aclaraciones o ampliar ciertos puntos y, por supuesto, hacer preguntas.

Es importante la claridad al hablar, expresarse de forma específica y, en lo posible, breve. Las **buenas preguntas** generan buenas respuestas y así tendremos una comunicación efectiva. Con esto, no estamos diciendo que no hagamos cualquier pregunta que deseemos, sino que prioricemos las preguntas importantes y las formulemos lo mejor posible.

Ahora, veamos a modo de ejemplo algo que podríamos llamar *¡un mal día en la vida de la persona haciendo pruebas!* Imaginemos que no contamos con el ambiente de pruebas para comenzar a trabajar y eso recortará las horas disponibles para *testing*, ya que perdemos días que no se recuperarán. ¿Qué pasa si quiero pedir explicaciones? Esperamos a estar a solas con nuestra contraparte. ¿Qué pasa si quiero criticar o estoy molesto? Esperamos a otro día para iniciar una conversación. Luego, conversamos haciendo foco en la situación y no en las personas. Este ejemplo es también para mostrar que no se deben tomar a título personal decisiones ajenas. **Manejar las quejas** es una habilidad, evitarlas ayuda a comunicar mejor y, además, muestra que somos capaces de adaptarnos al cambio.

Además de la habilidad de comunicación oral, es indispensable mejorar nuestra **capacidad de comunicación escrita**, lo que incluye la capacidad de redactar

informes claros, escribir correctamente un correo electrónico, redactar reportes de defectos que sean útiles, etc.

Para cualquier comunicación escrita, deberíamos en principio tener la precaución de escribir sin faltas de ortografía. La escritura se mejora leyendo libros, no revistas, no alguna noticia del diario de vez en cuando, se mejora leyendo libros, ya que estos pasan por un proceso de editorial más estricto involucrando profesionales con conocimientos amplios en el uso del lenguaje y la comunicación escrita. Para mejorar más rápido, es bueno estudiar algunas reglas de ortografía básicas, uso de tildes, etc. Existen manuales que podemos tener a mano para consultas, como por ejemplo *Gramática y ortografía*, de Carmen Lepre (Ed. Santillana) e incluso revisar dudas en la página de la Real Academia Española⁵⁹ o en la de la Fundación del Español Urgente⁶⁰ cuando sea necesario.

Es conveniente leer varias veces un texto antes de enviarlo y, si se trata de un plan o informe, está bien aplicar *peer review*⁶¹ junto a un compañero. La revisión de documentos es una buena práctica interna, no solamente para revisar la ortografía, sino que hay que ver la estructura, o sea, cómo está organizado el informe, si se entiende con facilidad, que no haya ambigüedades, si está completo, etc.

Para el envío de correos electrónicos, hay que tener algunas consideraciones extra. Si el servidor de correo lo permite, no es mala idea tener activada la funcionalidad *Undo* (que trae Gmail, por ejemplo). Esta funcionalidad da un bonus de tiempo, por ejemplo, para asociar un archivo adjunto que olvidamos, para agregar o quitar un destinatario de correo o para revisar una vez más el texto que redactamos, etc. Como buenas prácticas, al escribir un correo destacamos evitar el “Responder a todos”. “Responder a todos” hace que aumente el “volumen-ruido” de *mails* que recibimos cuando muchas veces no somos receptores directos del mensaje. Además, suponiendo que buscamos una rápida respuesta a un problema, si enviamos a todos, el posible problema queda abierto a que lo resuelva quien quiera asumirlo. En ocasiones, el correo puede comenzar siendo grupal pero a

⁵⁹ Real Academia Española (RAE): <<http://www.rae.es>>

⁶⁰ Fundación del Español Urgente: <<https://www.fundeu.es/sobre-fundeurae/quienes-somos>>

⁶¹ El *peer review* es un proceso de revisión por pares en el que el trabajo de una persona es evaluado por un par, o sea, con conocimientos y habilidades similares. El objetivo es mejorar la calidad al permitir que colegas revisen y proporcionen comentarios constructivos antes de su publicación.

medida que los mensajes van y vuelven, puede ir cambiando el contenido. Si cambia el contenido, entonces, lo mejor será cambiar el asunto y enviar solo al receptor o receptores necesarios.

Algo más a comentar de la comunicación escrita es que tiene ciertas limitaciones, por ejemplo, no se transmite el tono con que se dicen las cosas y muchas veces da lugar a malinterpretaciones. Es recomendable escribir lo más claro posible, buscando reducir la ambigüedad, ya que puede dar lugar a distintas interpretaciones.

Por último, hay que tener en cuenta aquellas situaciones en que hay que comunicar algo urgente. En ese caso, además de enviarlo por correo para que quede registro, también se utiliza el teléfono o se trata el tema personalmente. Lo mismo aplica para cuando estamos realizando pruebas y encontramos un incidente crítico, si es realmente grave, está bien avisar directamente y no solo reportarlo en un gestor de incidentes.

Hay otras consideraciones a tener en cuenta al reportar incidentes para lograr ser más efectivos en nuestra comunicación. Por ejemplo, incluir un paso a paso claro y fácil de seguir por quien lo lea. Puede ocurrir que estemos trabajando en un proyecto en el cual el equipo de desarrollo conozca profundamente el producto y, entonces, puede parecernos que no se necesitan detalles, aun así, siempre deberíamos incluir los pasos para reproducir un *bug*. Para un buen reporte intenta también incluir las condiciones del ambiente de *testing*, en qué navegador y en qué versión de la aplicación estábamos trabajando, los datos de prueba utilizados, así como capturas de pantalla.

Todos los incidentes que encontramos se reportan, los graves, los menores, los que no estamos seguros si son *bugs*. Por lo tanto, y ya que todo se reporta, es importante ser organizados y priorizar los incidentes, reportando los más importantes primero. Con esto no queremos decir que se reportan los *bugs major* primero y los demás se dejan atrás, sino que vamos reportando de acuerdo al impacto que tiene en el negocio. Intentemos hacer *testing orientado a la satisfacción del cliente*, por llamarlo de algún modo. Si encontramos un *bug* muy grave, pero en una funcionalidad que muy rara vez se usa, lo podemos reportar después.

En otra línea de pensamiento, quisiéramos compartir algunos tipos de frases, afirmaciones o modismos que hemos visto que se repiten de forma sistemática en nuestra industria, y creemos que deberían evitarse para lograr una comunicación más efectiva:

- **Evitar tomar más responsabilidades de las que nos corresponden.** Por ejemplo, es muy común que *testers* digan: “Está listo para pasar a producción”. Esta decisión no es nuestra, con lo cual sería mejor limitarnos a dar información para que a quien le corresponda tome la decisión, en base a la información que le proporcionamos y a otras cuestiones de contexto que podrá tener en consideración.
- **Evitar asegurar cosas que no sabemos por ciertas.** Por ejemplo: “El sistema no tiene *bugs*”. Como ya vimos al inicio de este libro, no se puede demostrar la ausencia de *bugs*. Una alternativa más adecuada a la realidad sería decir que “con las pruebas que realizamos no fuimos capaces de encontrar incidentes que consideremos de gravedad” o algo similar. Otro ejemplo muy común relacionado a pruebas de *performance* es: “El sistema soporta la carga de usuarios esperada”, cuando lo más adecuado es: “Con las pruebas que ejecutamos, no pudimos encontrar un problema que evite que el sistema soporte la carga esperada”. Claro que no vamos a hablar tan complicado, pero no está de más tener esto en cuenta para analizar cómo lo expresamos y cómo lo conceptualizamos.
- **Evitar hablar en términos absolutos.** *Nunca digas nunca, jamás.* Hay que ser cuidadosos con dar afirmaciones que generalicen algo en base a unas pocas observaciones. Por ejemplo: “A **todos** los usuarios les falta esa opción en el menú”, “siempre que entro al sistema me pasa tal cosa”. Lo peligroso de esas afirmaciones reside en los términos absolutos. Quizá nosotros vimos un comportamiento en un elemento, revisamos dos o tres y comprobamos que también sucede, pero eso no nos habilita a afirmar que todos tendrán ese mismo comportamiento. Otras palabras que representan el mismo tipo de potencial problema: *todos, absolutamente, siempre, completo, completamente, incesante, incesantemente, definitivamente, entero, nunca, cada, cada uno, cada cosa, lleno, deber, nunca, nada, totalmente, totalidad.* Nuestra sugerencia es que, si vamos a expresar algo con alguna de estas palabras, lo pensemos, y si recibimos algo de información con estas palabras, lo relativicemos.

- **Tener cuidado con los términos cuantificadores.** De manera similar, los términos cuantificadores, como por ejemplo, *muchos*, *algunos*, *pocos*, *la mayoría*, *la minoría*, etc. también representan algo que vale la pena revisar. En estos casos, sería bueno pasar de algo que parece una opinión a algo que muestre un dato. Por ejemplo, en lugar de decir: “En esta versión hay muchos errores”, podríamos decir: “En esta versión encontramos 12 errores críticos y 3 bloqueantes”. Otro caso similar es cuando se dice: “Los tiempos de respuesta bajaron considerablemente”. En cierta forma, esta frase parece no aportar información, sino tan solo una sensación o una percepción subjetiva. Quedaría mucho más claro si se lleva a algo como: “Los tiempos de respuesta bajaron de 4,3 segundos a 2,1” o “bajaron un 52%”.

Sensibilidad por la calidad

La sensibilidad por la calidad es una habilidad fundamental para el rol de *tester*. Al igual que algunas personas son más perceptivas a ciertos estímulos visuales (logran distinguir una gama más amplia de colores) o auditivos (identifican en qué nota está cada sonido que escuchan), hay quienes tienen la capacidad de percibir y valorar la calidad de un producto de manera más profunda.

Esto implica empatizar de mejor manera con los usuarios sin perder el foco de negocio. Al desarrollar esta habilidad, podremos identificar aspectos que van más allá de la funcionalidad del sistema, comprender la importancia de la experiencia del usuario, la integridad del código y otros factores de calidad como la usabilidad, seguridad, *performance*, etc.

Al ser capaces de percibir y evaluar la calidad de manera más refinada, se logra contribuir significativamente a la detección temprana de problemas y a la mejora continua de los procesos. De esta forma, no solo estaremos contribuyendo a que el producto final cumpla con la calidad esperada, sino que el equipo logrará cada vez mejores resultados.

Escepticismo y pensamiento crítico

Nos gusta tratar estas dos ideas juntas, ya que el *tester* debería contar con cierto escepticismo como para no caer en “creo que va a funcionar”, y la forma de lograrlo es cuestionar, dudar, experimentar y no creer hasta no ver.

El *pensamiento crítico* puede definirse como la acción de pensar sobre nuestra forma de pensar para evitar ser engañados. El cerebro está programado para trabajar poco, para ahorrar energía, entonces, se tiende a automatizar ciertas respuestas, ciertos mecanismos de razonamiento. Muchas veces razonamos sobre ciertas asunciones, tomando ciertas cosas por sentadas. Ejercitar el pensamiento crítico implica ir en contra de este modo de operar del cerebro: intentar evitar el reflejo o reacción automática y parar a reflexionar, intentar analizar lo que asumimos, y ver si podemos cambiar la forma de razonamiento en base a cambiar esas asunciones. Cuestionarnos las asunciones, cuestionar el razonamiento, cuestionar el problema, cuestionar la forma en que razonamos. Así, evitaremos que otros “nos engañen”, que el problema “nos engañe” o que nuestro propio cerebro “nos engañe”.

Al *pensamiento crítico* se lo asocia también a esa mirada “destruktiva” que siempre se le asocia al *testing*. En realidad, se trata de una mente que duda, que necesita pruebas y que va a pensar en situaciones que salen de la norma o el esquema que venía pensando desarrollo. Esto es parte del valor que brinda el *tester*.

Quizá un aspecto que nos gustaría que se considere es que por mantener esta visión objetiva, basada en datos, con mirada crítica y dudando de todo, no deberíamos perder el entusiasmo ni apuntar a quitárselo al resto del equipo. Una cosa es ser escéptico y crítico y otra es ser negacionista, negativo, pesimista, desconsiderado o poco compasivo al brindar *feedback*. Necesitamos encontrar un balance sano entre escepticismo y entusiasmo.

Conocimiento del negocio

Si bien para realizar tareas de *testing* con éxito no es indispensable ser un experto en el negocio, es sin dudas un *plus* acercarnos a los usuarios o al cliente, conocer cómo trabajan, preocuparnos por conocer los servicios de la empresa, conocer los productos, investigar productos similares que sean la competencia, etc. Con esta base, podemos aportar otras características a nuestro trabajo, no solamente reportar defectos, sino que podremos identificar las fortalezas y debilidades de un sistema e incluso sugerir funcionalidades que puedan mejorarlo.

Este es un tema que genera mucha controversia en el mundo del *testing*; creemos que es porque existe cierto miedo a que vengan personas de otras áreas a “robar” el trabajo de los informáticos. Hemos llegado a escuchar incluso justificaciones de que los informáticos siempre vamos a tener más capacidad de encontrar más incidentes. Estamos en desacuerdo con esto. Ambos perfiles son necesarios y, si una misma persona los combina, mucho mejor, pero un equipo multidisciplinario es lo ideal también. El ejemplo típico para mostrar esto es el de quien desarrolla un sistema financiero. Para formar un buen equipo de pruebas, serán necesarios expertos en tecnología, para probar los aspectos más técnicos, pero igual de importante serán los que tienen conocimientos profundos en finanzas y puedan identificar problemas del sistema más relacionados con el negocio, reglamentación, cálculos, etc.

Independencia y proactividad

Tener la capacidad de trabajar de manera independiente, sin supervisión directa es una característica particularmente importante. No solo se trata de disfrutar de la independencia, sino que hay que respetar la confianza que han depositado en nosotros y asumir que tenemos una responsabilidad entre manos.

Muchas veces en nuestro trabajo es necesario tomar acción y dar respuestas rápidamente y está bien animarse, tener la iniciativa de resolver los problemas que puedan surgir. Esto se asocia a la idea de proactividad, esa capacidad de

anticiparnos a las situaciones, tomar la iniciativa y actuar de forma temprana en lugar de simplemente reaccionar a eventos o circunstancias.

Está en nosotros desarrollar esta habilidad de ser autónomos, reconocer hasta dónde podemos contestar sin aprobación y qué temas no pueden seguir adelante sin que nos den un *OK* previo. Este es el tipo de habilidades que se espera que vayan mejorando a medida que la persona vaya ganando experiencia y *seniority*.

Manejo de la frustración

Esta es una habilidad crucial para un *tester*, ya que necesitaremos mantener la calma y la objetividad frente a situaciones adversas a las que les aseguramos se van a enfrentar a diario durante el proceso de prueba. “Toparnos” con algo que no funciona, con un ambiente que no es estable, con falta de documentación o de alguien que pueda ayudarnos son algunos de los primeros ejemplos que se nos ocurren. Será fundamental que, incluso en estas situaciones, seamos capaces de mantener la cordura, ser perseverantes y continuar en nuestra búsqueda por la resolución de problemas y persecución de la calidad. Deberemos encontrar las estrategias que nos permitan mantener la concentración y no dejar que la frustración afecte negativamente la calidad de nuestro trabajo.

¿Escucharon hablar de “el karma del *tester*”? La vida es un círculo, todo lo que va luego vuelve, todo termina en equilibrio al final⁶². Tomando lo que dice Wikipedia sobre el Karma⁶³: “...generalmente el karma se interpreta como una «ley» cósmica de retribución, o de causa y efecto. Se refiere al concepto de «acción» entendido como aquello que causa el comienzo del ciclo de causa y efecto. ...El karma explica los dramas humanos como la reacción a las acciones buenas o malas realizadas en el pasado más o menos inmediato”. Esto es lo que puede llevar a pensar que, si existe algo como **el karma del *tester***, entonces, hay una explicación a por qué en nuestra vida cotidiana nos encontramos con tantos fallos en todo el *software*. Debe ser por nuestros actos como *testers*. Como hacemos tanto esfuerzo por encontrar problemas, incidentes, *bugs*, en el *software*, luego, en nuestra vida cotidiana, nos

⁶² Karma del *tester* <<https://www.federico-toledo.com/el-karma-del-tester>>

⁶³ Karma <<https://es.wikipedia.org/wiki/Karma>>

enfrentamos a problemas, incidentes, *bugs* en el *software* que usamos a diario. Esto, no hay otra, se debe al karma del *tester*. Si aún no están en el mundo de la calidad de *software*, trabajando como *testers*, quizá estén a tiempo de pensarlo. Luego de comenzar a trabajar en *testing*, ¡observarán *bugs* por todos lados! Dicho todo esto, ¡qué importante tener un buen manejo de la frustración!

Habilidades personales vs. técnicas

Es importante que entre nuestras habilidades esté prestar atención a los detalles, priorizar, pensar en pos de la excelencia del equipo. La capacidad de reconocer, planificar y saber organizarnos en el entorno de trabajo es fundamental, así como es necesaria una fuerte capacidad analítica y lógica en la personalidad del *tester*. Ahora bien, ¿es necesario saber programar?

Si la persona que hace *testing* debe tener conocimientos técnicos o no es un tema siempre abierto a debate. Según una polémica presentación de James Whittaker (2011), el *Testing* Funcional tradicional estaría en vías de extinción y, por lo tanto, sería imprescindible orientarnos hacia un perfil técnico, o correríamos el riesgo de no continuar en carrera. Suena extremista, pero aun así no dejemos de reconocer algunos beneficios...

Tengamos en cuenta que las habilidades técnicas mejoran significativamente la comprensión de los sistemas bajo pruebas, son sin lugar a dudas **un valor agregado**. Es recomendado que el *tester* tenga conocimientos de programación, base de datos, etc. Dependiendo de los objetivos y del perfil de cada uno, el nivel técnico exigido será mayor o menor. Si se trata de una persona que realiza pruebas de *performance* o que automatiza pruebas, seguramente necesite codificar.

Otro beneficio de contar con habilidades técnicas es que puede facilitar la comunicación con el equipo de desarrollo. Ambas formaciones son complementarias. Se bromea a menudo con que el *tester* es el enemigo natural del desarrollador, pero hoy por hoy nuestra experiencia nos dice que somos un aliado esencial. Conocer algún lenguaje de programación, colaborar continuamente, comunicar claramente y con la prioridad y severidad acertadas son detalles que no solamente facilitarán cualquier comunicación, sino que favorecen el respeto entre profesionales.

Para quien desee especializarse orientándose hacia perfiles más técnicos, ¡sepan que todo se puede aprender! tanto estudiando como con la experiencia adquirida en los proyectos que vamos participando, pero ¿por dónde comenzamos?

Una pregunta muy común en *testers* que buscan aprender a automatizar como una forma de crecer, es ¿qué *skills* son necesarios para comenzar a automatizar? Según James Bach, en alguno de los artículos que escribió este *gurú*, no es necesario tener condiciones especiales para automatizar, lo ideal es que los mismos *testers*, que conocen de los requerimientos y del negocio de la aplicación, y que realizan el *Testing* Funcional tradicional, puedan encarar las tareas de automatización, evitando que recaiga en alguien que conozca de la herramienta nada más y solo un poco de la aplicación a testear. Es deseable que sean los mismos *testers* por varios motivos: no se genera competencia entre las pruebas manuales y las pruebas automatizadas, ayuda a asegurar la correcta elección de las pruebas que se realizarán de manera automatizada y, además, las herramientas de *Testing* Automatizado pueden servir no solo para automatizar casos de prueba, sino también para generar datos para los casos de prueba. Entonces, no hay motivo de preocupación para nosotros, los *testers*, ya que si se usan pruebas automatizadas no significa que nos vayan a querer sustituir.

Será importante conocer:

- la aplicación y el dominio de negocio
- la herramienta de automatización
- plataforma con la que se trabaja (para conocer los problemas técnicos típicos)
- *testing* (técnicas de generación de casos de prueba)

Cada *skill* va a aportar valor en distintos sentidos, hará que nuestro perfil “se mueva” en las distintas zonas del conocimiento planteadas en la siguiente figura. Claramente, mientras más nos acerquemos al centro, más capacidad vamos a tener de aportar valor al desarrollo de un producto, pero podríamos querer especializarnos en una de esas áreas o en alguna intersección en especial.



Conocimientos requeridos en *testing*

Por último, tengan en cuenta que la automatización de pruebas **no** es algo que se pueda hacer en los tiempos libres. Aconsejamos que, de manera paralela al trabajo manual, puedan comenzar con la capacitación en la herramienta que se vaya a utilizar, así como también leer material de metodología y experiencias de pruebas automatizadas en general.

Pasión y motivación

Como último punto de este capítulo, queríamos hablar de la **actitud** necesaria para ser mejores profesionales y poder crear día a día una mejor versión de nosotros mismos.

Amar nuestra profesión es la clave. Es fundamental tener verdadera pasión por lo que se hace (en esta y en cualquier profesión), es lo que te hará llegar con una actitud positiva al trabajo y dispuesto a hacer cuanto sea necesario para completar tus tareas, en lugar de estar contando los minutos para salir, con angustia por tener que cumplir, por obligación.

La pasión **convence**, ayuda a que podamos resolver problemas positivamente, a transformarlos en oportunidades y desafíos, a estar enfocados siempre en la solución, a pensar, hablar y actuar con optimismo.

La pasión **contagia**, ayuda a compartir conocimientos en lugar de competir. El éxito no tiene que ver con ser mejor que los demás, el éxito tiene que ver con crear éxito en otras personas. La pasión ayuda a crear esa sensación de equipo.

Lógicamente, algunos días podremos estar desanimados, toda carrera tiene altos y bajos, pero ¿qué podemos hacer para que no nos gane el desánimo? En nuestro trabajo hay picos de intenso trabajo y momentos más tranquilos, y en todos podemos aprender continuamente. Puedes buscar nuevas herramientas para probar, aplicar una técnica nueva, pensar qué es lo que te gusta de lo que ya estás haciendo y dedicarle más tiempo, o plantearte una nueva meta (alcanzable y concreta). También habrá momentos en que haya que afrontar situaciones que nos hagan “salir de la zona de confort”, por ejemplo, si siempre hemos trabajado como *tester* manual y comenzamos a automatizar o si no nos gusta hablar en público y se nos da la oportunidad de dar una presentación.

Tarde o temprano hay que “dar el salto” y **desarrollar la capacidad de hacer algo diferente**. Todo nos ayudará a realizar mejor nuestra profesión.

Comentarios finales del capítulo

¿Con esto ya estamos listos para probar lo que sea? Bueno, tenemos una base, pero siempre es importante investigar más técnicas y tenerlas presentes, como para que en cada nueva situación tengamos la capacidad de seleccionar la más adecuada para el contexto que haya que probar (a modo de repaso y, muy en alto nivel: la de Tablas de Decisión es útil cuando hay reglas de negocio definiendo cierta lógica; la de Máquinas de Estado, para cuando tenemos entidades que cambian el comportamiento según su estado; la de Matriz CRUD, para ciclo de vida de entidades; la de Grafos Causa-Efecto, para cuando haya reglas del tipo *if-then-else*).

¿Cómo seguir?

- Practicar estas técnicas con casos reales, con aplicaciones que usen a diario, que prueben a diario.
- Investigar otras técnicas. Existen cientos de fuentes de donde obtener casos con ejemplos, experiencias prácticas.
- *Pair testing*: trabajar en conjunto con otro *tester* e intercambiar técnicas, mostrar cómo cada uno aplica cada técnica y discutir sobre cuál es la mejor forma para lograr los mejores resultados.

Estaremos encantados también de recibir sus inquietudes o comentarios sobre estas técnicas u otras que sean de su agrado (una forma es comunicándote al contacto de Abstracta: <hello@abstracta.us>. También queda abierta la invitación a conectar por Linkedin: <<https://www.linkedin.com/in/federicotoledo>>

La ciencia se compone de errores, que, a su vez, son los pasos hacia la verdad.

Julio Verne

CIERRE

Brené Brown (2010) dice algo así como “uno aprende a nadar nadando, uno aprende a tener coraje teniendo coraje”. Podríamos seguir escribiendo capítulos y capítulos sobre *testing* para ofrecer más temas para aprender, pero la verdadera forma de aprender es ir y hacer. Estamos muy agradecidos por el tiempo dedicado en leernos, pero ahora, para realmente mejorar las *skills* de *tester*, hay que cerrar el libro y ponerse a probar. ¡Esto recién empieza!

Lista de referencias:

Bach, J. (2003) Exploratory testing explained. Developsense.

Recuperado de <<http://satisfice.us/articles/et-article.pdf>>

Bach, J. (2007) What is test automation? Satisfice. Recuperado de

<<http://www.satisfice.com/blog/archives/118>>

Bach, J. (2024). Testing and checking refined. Satisfice. Recuperado de

<<http://www.satisfice.com/blog/archives/856>>

Bolton, M. (2009) Testing vs. checking. Developsense. Recuperado de

<<http://www.developsense.com/blog/2009/08/testing-vs-checking>>

Bolton, M.(2010). Testers: Get out of the quality assurance business.

Developsense. Recuperado de

<<http://www.developsense.com/blog/2010/05/testers-get-out-of-the-quality-assurance-business>>

Brown, B. (2010). Recuperado de <<https://daretolead.brenebrown.com>>

Capers, J. (1996) Applied software measurement: assuring productivity and quality. Recuperado de

<https://www.goodreads.com/book/show/2110359.Applied_Software_Measurement>

Cohn, M. (2009). Succeeding with agile. Recuperado de

<<https://www.goodreads.com/book/show/6707987-succeeding-with-agile>>

Daft Punk (2001). Harder, better, faster, stronger, en Discovery. Bercy, París.

Fewster, M., Graham, D (1999). Software test automation: effective use of test execution tools. ACM Press Books. Recuperado de

<<https://ebin.pub/software-test-automation-effective-use-of-test-execution-tools-illustrated-edition-9780201331400-0-201-33140-3.html>>

Fowler, M. (2018). Practical test pyramid. Recuperado de
<<https://martinfowler.com/articles/practical-test-pyramid.html>>

Galeano, E. (1989). El libro de los abrazos. Siglo XXI Editores.
Recuperado de <https://www.goodreads.com/book/show/1073100.El_libro_de_los_abrazos>

Google (2016). The need for mobile speed. Recuperado de
<<https://blog.google/products/admanager/the-need-for-mobile-speed>>

Hendrickson, E. (2012) Explore it! Pragmatic Bookshelf.
Recuperado de <<https://www.goodreads.com/book/show/15980494-explore-it>>

**Institute of Electrical & Electronics Engineers, Standard 610 (1990).
Standard glossary of software engineering terminology.** Recuperado de
<<https://ieeexplore.ieee.org/document/182763>>

**Kaner C. (2003). What is a good test case? Improving the education of
software testers.** Recuperado de <https://kaner.com/?page_id=7>

**Kaner, C. (2006). Exploratory testing after 23 years, Cem Kaner.
Conference of the Association for Software Testing.** Recuperado de
<https://kaner.com/?page_id=7>

**Kaner, C, Bach, J., Pettichord, B. (2001). Lessons learned in software
testing.** Recuperado de
<https://www.goodreads.com/book/show/599997.Lessons_Learned_in_Software_Testing>

Keay, J. (1994) The permanent book of exploration. Carroll & Graf Pub.
Recuperado de
<https://www.goodreads.com/book/show/1714290.The_Permanent_Book_Of_Exploration>

Marick,B.: Test idea. Testing Foundation. Recuperado de
<<http://www.exampler.com/testing-com/tools.html>>

Meier, J., Farre, C., Bansode, P., Barber, S., Rea, D.: Performance testing guidance for web applications: patterns & practices. Microsoft Press (2007).

<<https://www.goodreads.com/book/show/35129632-performance-testing-guidance-for-web-applications>>

Orejas, F. (26 de julio de 2012), ¿Es posible construir un software que no falle?, El País, Madrid. Recuperado de

<<http://blogs.elpais.com/turing/2012/07/es-posible-construir-software-que-no-falle.html>>

Pressman, R. (2010). Ingeniería de software, un enfoque práctico. Ed. McGraw-Hill. México. Recuperado de

<https://www.goodreads.com/book/show/142783.Software_Engineering>

Pyhäjärvi, M. (2017) Exploratory testing. Leanpub. Recuperado de

<<https://www.goodreads.com/book/show/34120133-exploratory-testing>>

Rice, R. (2003). Surviving the top ten challenges of software test automation. Semantic Scholar. Recuperado de

<<https://www.semanticscholar.org/paper/Surviving-the-Top-Ten-Challenges-of-Software-Test-Rice/8fe022ef115fd69cfb7d1c79f7f9a058a29f5d50>>

Rowe, S. (2020). Recuperado de

<<http://blogs.msdn.com/steverowe/archive/2007/12/19/what-is-test-automation.aspx>>

Saint-Exupéry (1943), A. de, El Principito, Publicaciones y Ediciones Salamandra, S.A.: Barcelona.

Toledo, F. (2017) Continuous integration, continuous delivery y continuous deployment. Federico Toledo. Recuperado de

<<https://www.federico-toledo.com/diferencia-entre-continuous-integration-delivery-y-deployment>>

Toledo, F. (2021). Low code for test automation. Blog de Abstracta.

Recuperado de <<https://abstracta.us/blog/software-testing/low-code-for-test-automation>>

Whittaker, J. (2011). STARwest Keynote. Recuperado de

<<http://www.testthisblog.com/2011/10/james-whittakers-starwest-keynote.html>>

Weinberg, G. (2008). Perfect software: And other illusions about testing. Recuperado de <<https://www.goodreads.com/book/show/4060560-perfect-software>>

Weinberg,G,(2008). Perfect software: And other illusions about testing. Recuperado de <<https://www.goodreads.com/book/show/4060560-perfect-software>>

Zallar, K. Practical experience in automated software testing. Methods and tools. Recuperado de <<http://www.methodsandtools.com/archive/archive.php?id=33>>

Reseña

“*Introducción a las pruebas de sistemas de información*” de Federico Toledo es una obra pionera en español que aborda de manera integral y práctica los diversos tipos de *testing*. El libro es una guía completa que cubre desde conceptos básicos hasta técnicas avanzadas de diseño de pruebas funcionales, automatizadas y de *performance*, así como las habilidades esenciales para los *testers*. Con un enfoque dinámico y práctico, proporciona herramientas y metodologías efectivas para enfrentar los desafíos del *testing* en el entorno profesional, destacando la importancia de integrar aspectos técnicos y humanos en el proceso de *testing*. Fede Toledo combina su destacada trayectoria en el campo del *testing* con un estilo claro y accesible, facilitando la rápida incorporación de conocimientos para el lector. La obra está respaldada por la experiencia del equipo de Abstracta, ofreciendo una perspectiva enriquecida y actualizada sobre las mejores prácticas en *testing*. Dirigido a profesionales del *testing*, este libro es una referencia esencial que no solo busca mejorar las habilidades técnicas de los *testers*, sino también potenciar su capacidad de comunicación y su comprensión integral de los sistemas de información. Es un libro que he recomendado ampliamente a todos los *testers* hispanos que he conocido por su gran cobertura, utilidad práctica, referencias e idioma.

Antonio Jiménez, *Performance Engineer*

Durante estos diez años, “*Introducción a las pruebas de sistemas de información*” ha logrado mucho más que simplemente enseñar sobre *testing*. Ha sido fundamental en el fortalecimiento de nuestra comunidad, porque aborda de manera integral y práctica aspectos clave del *testing*, permitiendo que muchas personas se adentren con facilidad en el mundo de las pruebas de *software*. Fede, gracias por ofrecernos una guía tan completa donde adicionalmente aboradas aspectos profundamente humanos que subrayan la importancia de la excelencia personal en esta profesión.

Mercedes Quintero, CoCEO Regional Abstracta

ISBN: 978-9915-42-634-1



abstracta.us