Introducción a las **Pruebas de Sistemas de Información**

FEDERICO TOLEDO

Introducción a las Pruebas de Sistemas de Información

Autor: Federico Toledo Rodríguez

Co-autores: Andrés Curcio y Giulliana Scuoteguazza

Colaboradores: todo el equipo de Abstracta

Abstracta, Montevideo, Uruguay, 2014

PRÓLOGO

Reseña

"Introducción a las Pruebas de Sistemas de Información" es una de las primeras publicaciones en español sobre aspectos prácticos de testing. En el mismo se abordan todas las dimensiones del testing, mostrando su propósito y beneficio. Integrando técnicas modernas de forma dinámica y práctica le permite al lector realizar una recorrida nutrida y clara sobre el diseño de pruebas, pruebas automatizadas, pruebas de performance y aspectos humanos que debe desarrollar un buen tester.

Su autor, el ingeniero Federico Toledo, PHD en Informática en la Universidad de Castilla-La Mancha, es un especialista en testing y cuenta con una destacada trayectoria privada, institucional y académica en Uruguay –su país de nacimiento– y Europa. Es socio fundador e integrante permanente del directorio de ABSTRACTA, empresa que provee productos y servicios de testing a compañías en Estados Unidos y América Latina, y es un blogger frecuente de artículos especializados en testing en lengua castellana. Federico, destaca el carácter integral de su obra, en cuanto a conceptos y metodología y rescata la invalorable contribución que realizaron todos los integrantes de Abstracta, en especial Andrés Curcio y Giulliana Scuoteguazza, quienes colaboraron fervientemente para que el libro esté hoy en sus manos.

El público objetivo de "Introducción a las Pruebas de Sistemas de Información" son las personas que trabajan cotidianamente haciendo testing y se enfrentan a un sinnúmero de desafíos y dificultades. Ellos contarán a partir de hoy con una guía de apoyo para enfrentar distintas actividades que realizan, especialmente las relativas a técnicas de diseño de testing, testing automatizado y de performance.

"Introducción a las Pruebas de Sistemas de Información" está escrito en un lenguaje llano y ameno, y fue pensado y redactado para que su lectura permita la rápida incorporación de conceptos, técnicas y nuevas habilidades para sus lectores. El aprender y disfrutar aprendiendo fue uno de los intereses mayores que sostuvo Federico a la hora de estructurar y dar contenido a su obra.

Temas

Introducción a las Pruebas Funcionales

Comenzando 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 termina utilizando casi en forma inconsciente, tales como: partición en clases de equivalencia, valores límites, etc.

Técnicas de Diseño de Pruebas Funcionales

Habiendo visto las básicas de las pruebas funcionales podremos seguir luego 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

Es una técnica bastante avanzada que consiste en diseñar y ejecutar al mismo tiempo. Veremos que existen escenarios especiales en las que nos va a resultar principalmente útil, y también cómo podríamos aplicar la técnica en forma práctica.

Pruebas Automatizadas

Veremos cómo el testing automático 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 veremos cómo automatizar pruebas con el fin de simular múltiples usuarios concurrentes y así poder analizar el rendimiento de la aplicación bajo pruebas, sus tiempos de respuesta y el consumo de recursos.

Habilidades de un Tester

Porque no todo en la vida son solo técnicas y tecnologías, también abordaremos algunos puntos más relacionados a los aspectos humanos que todo tester también debe preocuparse por desarrollar, especialmente relacionados a la capacidad de comunicar.

Síntesis final

Auguramos que “Introducción a las Pruebas de Sistemas de Información” sea una excelente experiencia de lectura y aprendizaje que efectivamente eleven al lector a un mayor nivel de conocimiento conceptual y práctico. ¡Ahora, a disfrutarlo!

Lic. Gonzalo Acuña

AGRADECIMIENTOS

Este es un libro con un carácter divulgativo. Nada de lo que se presenta aquí es nuestro, y al mismo tiempo todo lo presentado aquí es nuestro, pues lo hicimos parte de nuestra cultura de testing, adaptándolo según nuestros criterios y necesidades.

En este libro queremos plasmar conceptos, buenas prácticas, técnicas y metodologías que hemos investigado y que nos han resultado útiles, y creemos que les pueden resultar útiles también.

Como se trata de nuestro primer libro tampoco estamos siendo ambiciosos de querer tener algo completamente extenso, sino que priorizamos los conceptos, técnicas y métodos que nos resultan de mayor prioridad para entregar en esta primera versión (vaya que hasta aquí estamos aplicando conceptos de testing: no exhaustivo, priorizando lo importante para la liberación del producto).

Los que nos conocen, los que han visto charlas nuestras, o quienes leen nuestro blog (blog.abstracta.com.uy) se darán cuenta que muchas de estas cosas las hemos nombrado ya en otras ocasiones. Aquí las estamos organizando y comentando con más formalidad (bueno, quizá no tanta formalidad, veremos).

El agradecimiento va a todos los que nos han hecho crecer trabajando con nosotros, tanto los integrantes del equipo de Abstracta, como colegas y amigos de empresas partners y clientes. A todos ellos, ¡muchas gracias!

CONTENIDO

INTRODUCCIÓN

¿ES POSIBLE CONSTRUIR UN SOFTWARE QUE NO FALLA?

Con motivo del año de Turing –celebrado en 2012– el diario El País de España publicó esta noticia muy interesante[1]. Comienza preguntándose si es posible construir sistemas de software que no fallen, ya que hoy en día estamos todos muy acostumbrados a que todos los sistemas fallan. Ya es parte del día a día verse afectado por la poca calidad de distintos sistemas de los que dependemos. El tema es que existen también miles de ejemplos de errores que han producido grandes tragedias, muertes, accidentes, pérdidas de miles de dólares, etc.

El artículo comienza dando ejemplos de catástrofes (típico en todo profeta del testing) y luego plantea un ejemplo real que da esperanzas de 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 código y la verificación automática de estos sistemas generados.

El ejemplo planteado es el del sistema informático de la línea 14 del metro de París[2]. Esta línea es 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 2007, un promedio de 450.000 pasajeros toman esta línea en un día laboral). Hoy en día hay dos líneas de metro que funcionan con el mismo sistema en París: la línea 14 y la línea 1.

---

[1] Nota del diario El País: http://blogs.elpais.com/turing/2012/07/es-posible-construir-software-que-no-falle.html

[2] Metro de París nº 14: http://es.wikipedia.org/wiki/L%C3%ADnea_14_del_Metro_de_Par%C3%ADs

Según cuenta el artículo de El País, el sistema tiene 86.000 líneas de código ADA (definitivamente no es ningún "*hello world*"[3]). Fue puesto en funcionamiento en 1998 y hasta 2007 no presentó ningún fallo.

¿A qué queremos llegar con esto? A que es posible pensar en que somos capaces de desarrollar software suficientemente bueno. Seguramente este sistema tenga errores, pero no se han manifestado. Eso es lo importante. ¿Y cómo lo garantizamos? Obviamente, con un testing suficientemente bueno. Para esto tenemos que ser capaces de verificar los comportamientos del sistema que son más típicos, que pueden afectar más a los usuarios, que son más importantes para el negocio. Obviamente esto estará limitado o potenciado por otros factores más relacionados al mercado, pero nosotros nos centraremos más en cómo asegurar la calidad con el menor costo posible, dejando esas otras discusiones un poco al margen.

Esta visión es un poco más feliz que la que siempre escuchamos de Dijkstra[4], 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 al testing. Lo importante es hacerlo en una forma que aporte valor y nos permita acercarnos a ese *software perfecto* o mejor dicho: de calidad.

[3] Con "Hello world" nos referimos al primer programa básico que generalmente se implementa cuando alguien experimenta con una nueva tecnología o lenguaje de programación.

[4] Edsger Dijkstra: http://en.wikipedia.org/wiki/Edsger_W._Dijkstra

La Calidad, El Testing y sus Objetivos

El *objetivo* del *testing* es aportar *calidad*. Todo en una frase. Aportar a la calidad del producto que se está verificando (esto lo diremos cuantas veces sea necesario). 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? A quién no le ha pasado de que en cada curso, seminario, tutorial, etc., relacionado con testing nos hayan hecho esta pregunta, y cada vez que nos la hacen nos tenemos que poner a pensar nuevamente como si fuese la primera vez que pensamos en esa cuestión.

Siempre se vuelve difícil llegar a un consenso sobre el tema. En lo que todo el mundo está de acuerdo es que es un concepto muy subjetivo.

Podríamos decir que es una **característica que nos permite comparar distintas cosas del mismo tipo**. Se define, o 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 mí es de calidad, tal vez para otra persona no. Algo que para mí hoy es de calidad, quizá el año próximo ya no lo sea. Complejo, ¿no?

Según Jerry Weinberg "la calidad es valor para una persona", lo cual fue extendido por Cem Kaner diciendo que "la calidad es valor para una persona a la que le interesa".

Relacionado con este tema existe una falacia conocida como "Falacia del Nirvana" a tener presente. Esta refiere al error lógico de querer comparar cosas reales con otras irreales o idealizadas[5]. Y lo peor de esto, es que se tiende a pensar que las opciones reales son malas por no ser perfectas como esas 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", que también la hemos escuchado como "lo perfecto es enemigo de lo bueno".

---

[5] Ver http://es.wikipedia.org/wiki/Falacia_del_Nirvana

Testing

Según Cem Kaner es una investigación técnica realizada para proveer información a los *stakeholders* sobre la calidad del producto o servicio bajo pruebas. Según Glenford Myers, software testing es un proceso diseñado para asegurar que el código hace lo que se supone que debe hacer y no hace lo que se supone que no debe. Según el autor, se trata de un proceso en el que se ejecuta un programa con el objetivo de encontrar errores.

Lo más importante: aportar calidad al software que se está verificando. ¿Cómo? Detectando posibles incidencias de diversa índole que pueda tener cuando esté en uso.

Objetivos

Y la forma en la que estaremos aportando calidad es principalmente buscando fallos[6]. Por supuesto. Digamos que si no encuentro fallo todo el costo del testing me lo pude haber ahorrado. ¿No? ¡Nooo! Porque si no encontramos fallos entonces tenemos más confianza en que los usuarios encontrarán menos problemas.

¿Se trata de encontrar la mayor cantidad de fallos posible? ¿Un tester que encuentra cien fallos en un día hizo mejor su trabajo que uno que apenas encontró 15? De ser así, la estrategia más efectiva sería centrarse en un módulo que esté más verde, que tenga errores más fáciles de encontrar, me centro en diseñar y ejecutar más pruebas para ese módulo, pues ahí encontraré más fallos.

¡NO! La idea es encontrar la mayor cantidad de fallos que más calidad le aporten al producto. O sea, los que al cliente más le van a molestar.

¡Ojo!, el objetivo no tiene por qué ser el mismo a lo largo del tiempo, pero sí es importante que se tenga claro en cada momento cuál es. Puede ser válido en cierto momento tener como objetivo del testing 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 los usuarios, entonces seguramente se enfoque el testing en los escenarios más usados por los clientes.

---

[6] 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., etc.

El Testing al Final del Proyecto

Scott Barber dice 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 tendremos actividades de testing para realizar, que aportarán en distintos aspectos de la calidad final del producto. Si dejamos para el final nos costará mucho más, más tiempo, más horas, y más costo de reparación, pues si encontramos un problema en la arquitectura del sistema cuando este ya se implementó, entonces los cambios serán más costosos de implementar.

Ahora, ¿por qué no lo hacemos así siempre? Acá hay un problema de raíz. Ya desde nuestra formación aprendemos un proceso un poco estricto, que se refleja bastante en la industria[7]:

- Primero matemáticas y materias formativas, 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 (en C++).
- Un lenguaje con más abstracción (Java).
- Sistemas operativos, arquitectura, y otras de la misma rama.
- Luego, un proceso pesado como lo es el RUP[8], poniéndolo en práctica en un grupo de 12 a 14 personas. Recién en estas materias se ve algo de testing.
- Para 4to o 5to año suelen haber electivas específicas de testing.

¿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

[7] Basado principalmente en nuestra experiencia en Uruguay, pero ya hemos conversado con distintos colegas en otros países –como en Argentina o España– y no escapa mucho de este esquema.

[8] RUP: http://es.wikipedia.org/wiki/RUP

debería ir de la mano con el desarrollo en sí. El testing debería ser un conjunto de distintas actividades que acompañan y retroalimentan al desarrollo desde un principio.

Vale destacar que es un proceso empírico, se basa en la experimentación, en donde se le brinda información sobre la calidad de un producto o servicio a alguien que está interesado en el mismo.

TESTING INDEPENDIENTE[9]

En cualquier proyecto existe un conflicto de intereses inherente que aparece cuando comienza la prueba. Se pide a la gente que construyó el software que lo pruebe. Esto no parece tener nada raro, pues quienes construyeron el software son los que mejor lo conocen. El problema es que, así como un arquitecto que está orgulloso de su edificio e intenta evitar que le encuentren defectos a su obra, los programadores tienen interés en demostrar que el programa está libre de errores y que funciona de acuerdo con las especificaciones del cliente, habiendo sido construido en los plazos adecuados, y con el presupuesto previsto.

Tal como lo ha dicho alguna vez Antoine de Saint-Exupery, "*Es mucho más difícil juzgarse a sí mismo que juzgar a los demás*".

Desde el punto de vista del programador (el constructor del software) el tester tiene un enfoque destructivo, pues intenta destruirlo (o al menos mostrar que está roto). Si bien es cierto que el enfoque del tester 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 la calidad del mismo.

Pero atención: el tester 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!

[9] Basado en un fragmento de "Ingeniería de Software, un enfoque práctico" de Pressman.

Estructura del Resto del Libro

En el resto del libro abordamos seis grandes temas, o secciones, relacionados a las pruebas:

- **Introducción a las Pruebas Funcionales**: comenzando 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 termina utilizando casi en forma inconsciente, tales como: partición en clases de equivalencia, valores límites, etc.
- **Técnicas de Diseño de Pruebas Funcionales:** habiendo visto las básicas de las pruebas funcionales podremos seguir luego 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 las pruebas ad-hoc, pero no se trata de no pensar o no diseñar pruebas, o no aplicar conocimientos en testing. Al contrario, es una técnica bastante avanzada que consiste en diseñar y ejecutar al mismo tiempo. Veremos que existen escenarios especiales en los que nos va a resultar principalmente útil, y también cómo podríamos aplicar la técnica en forma práctica.
- **Pruebas Automatizadas**: veremos cómo el testing automático 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 veremos cómo automatizar pruebas con el fin de simular múltiples usuarios concurrentes y así poder analizar el rendimiento de la aplicación bajo pruebas, sus tiempos de respuesta y el consumo de recursos.
- **Habilidades de un Tester**: porque no todo en la vida son solo técnicas y tecnologías, también queremos tocar algunos puntos más relacionados a los aspectos humanos que todo tester también debe preocuparse 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 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 la idea es ver algunas de las tantas técnicas de diseño de casos de prueba y datos de prueba, que especialmente aplican al probar sistemas de información con bases de datos, y para algunas situaciones en particular. 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 al tester que comienza a meterse en el mundo de las pruebas a seguir un método simple y efectivo; y a medida adquiere experiencia podrá ir adquiriendo más técnicas y mejorando sus habilidades con absoluta confianza.

Conceptos Introductorios

En este apartado comenzaremos dando definiciones básicas. Si estás leyendo el libro significa que te interesa el testing, y probablemente conozcas todas estas definiciones, pero si en algún momento necesitas repasar cualquier 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:

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 "La Idea de Prueba":

> *"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."*

Por último, una definición provista por Cem Kaner:

> *"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. Ésta debe incluir varios elementos en su descripción, y entre los que queremos destacar se encuentran:

- 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, nunca lo tengo claro. Supongo 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 definen 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).

Para verlo aún más simple, podríamos decir que la cobertura es como cuando barremos la casa. Siempre se me olvida el cuarto, eso es que en mi barrido no estoy cubriendo el cuarto. Mide la calidad de mi barrido, y a su vez me da una medida para saber cuándo tengo que terminar de barrer: cuando cubra 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 miré! Entonces, ¡ojo!, 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 me garantizará que está todo probado.

¿Entonces para qué me sirve?

- Medida de calidad de cómo barro
- Me indica cuándo parar de barrer
- Me sugiere qué más barrer

Unos criterios pueden ser más fuertes que otros, entonces el conocerlos me 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.

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 al programar y utilizar un "menor" en lugar de un "menor igual", etc. También existen técnicas más avanzadas como la que implica el uso de máquinas de estado, etc. 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 es 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.

Cada caso de prueba abstracto puede ser instanciado con distintos valores, por lo que tendrá, al momento de ser ejecutado (o diseñado a bajo nivel) un conjunto de casos de prueba específicos, donde se asigna un valor concreto a cada variable (tanto de entrada como de salida) de acuerdo a las propiedades y restricciones lógicas que tiene determinadas.

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 la nombremos en varias situaciones: pruebas dirigidas por datos, o del inglés bien conocido como *Data-Driven Testing*.

Esta es una técnica para construir casos de prueba basándose en los datos 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.

Por ejemplo y para que quede más claro, se podría automatizar el “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, y dar OK, y nos aparecería un mensaje que diga “Bienvenido”. Por otro lado tendríamos una fuente de datos en donde especificaríamos distintas combinaciones de nombres de usuario y password, y mensaje de respuesta esperado. El día de hoy se me ocurre poner un usuario válido, password válido y mensaje de bienvenida; mañana agrego otro que valida que ante un password inválido me da un mensaje de error correspondiente, y esto lo hago 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, alimentando la técnica, personalizándola de acuerdo a nuestros gustos, criterios y formas de pensar que se ajusten a la aplicación sobre la que estemos trabajando.

Primero que nada, vamos a seguir –siempre que sea posible– 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 Figura 1 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.

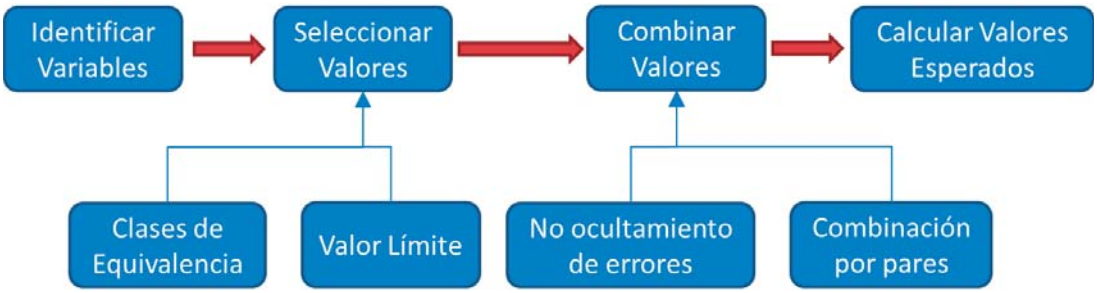
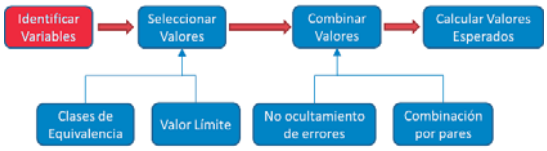


FIGURA 1 – PROCESO BÁSICO DE DISEÑO DE PRUEBAS

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 tendrán "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).
- Etc.

Existen muchas entradas de un sistema que tal vez no afectan 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, y no focalizar en otras pues esto nos hará diseñar y ejecutar 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 Figura 2 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.

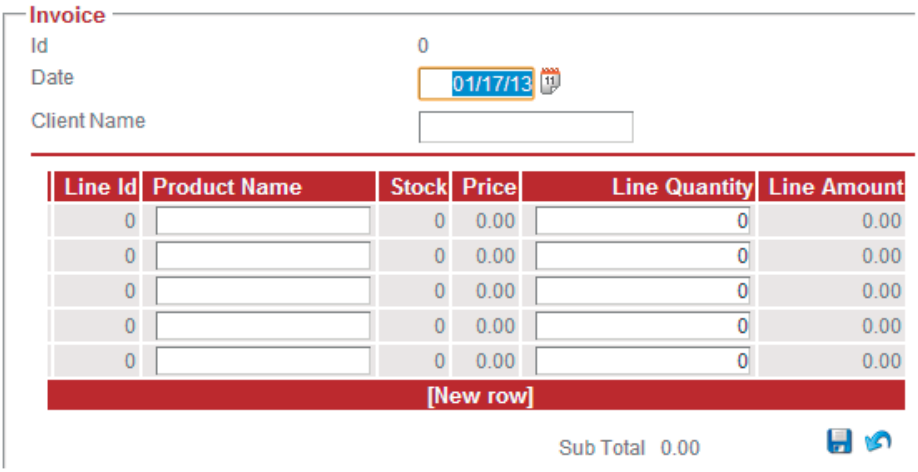


FIGURA 2 - PANTALLA PARA INGRESAR FACTURAS EN UN SISTEMA DE EJEMPLO

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 las 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* en la operación.

Analizando más la especificación del sistema, nos encontramos con que se registra el balance del 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 le asignarán en las pruebas. Para cada variable se decidirá utilizar distintos datos, los cuales sean interesantes desde el punto de vista del testing.

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 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, "Federico_Toledo" que es muy largo, por lo que debe dar error. En cambio, "Federico" y "Andrés" son ambas válidas, no son "significativamente diferentes", por lo que no 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 (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).
- 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 que si 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. 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.

Además, serán también interesantes los **valores límites** de las clases de equivalencia, es decir, para la clase 2 el valor 0 y el 17 y 18. 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 pueden 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 Figura 3.

FIGURA 3 - 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. 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. Los clientes son tratados distinto 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 Tabla 1.

Variable	Clases de equivalencia	Válida o inválida	Valores interesantes
Nombre	Hasta 30 Caracteres	Válida	“Federico”
	Más de 30	Inválida	String de 31 caracteres
	Vacía	Inválida	“”
Apellido	Hasta 30 Caracteres	Válida	“Toledo”
	Más de 30	Inválida	String de 31 caracteres
	Vacía	Inválida	“”
Edad	0<=x<18	Inválida	0, 1, 5, 17
	18<=x<=100	Válida	18, 30, 100, 101
	Negativos	Inválida	-1
	Caracteres no numéricos	Inválida	“asdf”
País	Cualquiera extranjero	Válida	España
	Local	Válida	Uruguay
Ciudad	Cualquiera	Válida	(cualquiera seleccionable)
Dirección	Hasta 100 Caracteres	Válida	“Bulevar Artigas, 1”
	Más de 100	Inválida	String de 101 caracteres
	Vacía	Inválida	“”
Balance			

TABLA 1 - DISEÑO DE DATOS DE PRUEBA PARA EL EJEMPLO

¡Ejercicio! Intente 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 o negativa o nula, y en cada caso será distinto el comportamiento del sistema. 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. 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 lo 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 = 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 sea todos los pares, y justamente lo que se hace es intentar utilizar todos los pares posibles. Imaginen que tenemos 3 variables para un caso de prueba sencillo, y se seleccionaron valores interesantes para cada variable como muestra el siguiente ejemplo (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, e-mail}
- 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). Tendríamos que diseñar pruebas para cubrir todos los pares posibles. Queda claro que intentar hacer esto manualmente sería una tarea muy costosa.

Existen muchas herramientas 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). Entre ellas destacamos una llamada CTweb (Combinatorial Testing Web *ctweb.abstracta.com.uy*), la cual implementa diferentes algoritmos para conseguir cubrir todos los pares de distintos conjuntos de datos de prueba (además de otras funcionalidades muy útiles).

Dentro de los distintos algoritmos existentes, el que nos resulta generalmente más útil es el llamado PROW (*pairwise with restrictions, order and weight*). Utilizando este algoritmo podemos excluir pares que nos resulten inválidos, e incluso darle distintas prioridades a los distintos pares, pues siempre será necesario repetir algunos pares para lograr todas las combinaciones, entonces quizá sea más interesante que se repita un par que otro, ya que esa combinación de datos ocurre más en la realidad que el resto.

Sigamos el ejemplo con la herramienta. Primero ingresamos al sistema y definimos las variables con sus valores interesantes, seleccionamos el algoritmo PROW y presionamos el botón “Execute” (ver Figura 4).

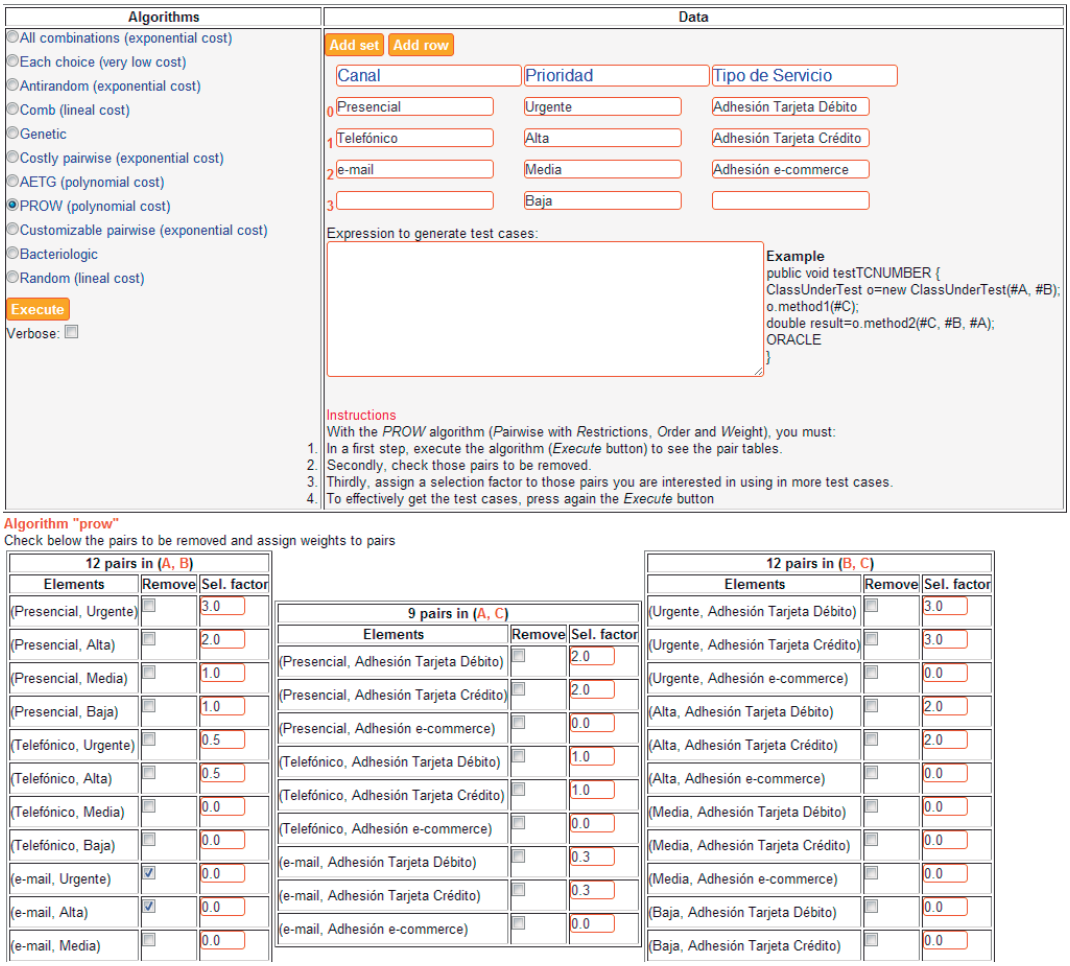


FIGURA 4 - EJEMPLO USANDO CTWEB

Enseguida nos mostrará los distintos pares existentes para los valores ingresados, tanto para que podamos indicar el peso de cada uno o si queremos remover alguno de esos pares por ser inválidos según las reglas del negocio. En el ejemplo ingresamos mayor valor al peso de los pares que son más importantes para la solicitud de servicios, y removimos los pares (e-mail, Urgente) y (e-mail, Alta), ya que no se permite hacer solicitudes por correo electrónico de prioridad alta y urgente.

Le damos nuevamente al botón “Execute” y nos aparece el resultado de las distintas combinaciones (ver Figura 5).

Algorithm "prow"

#	Results for a maximum of 36 combinations
1	{Presencial,Urgente,Adhesión Tarjeta Crédito}; sel. factor=8.0
2	{Presencial,Urgente,Adhesión Tarjeta Débito}; sel. factor=8.0
3	{Presencial,Alta,Adhesión Tarjeta Crédito}; sel. factor=6.0
4	{Telefónico,Alta,Adhesión Tarjeta Débito}; sel. factor=3.5
5	{Presencial,Baja,Adhesión Tarjeta Crédito}; sel. factor=3.0
6	{Presencial,Media,Adhesión Tarjeta Débito}; sel. factor=3.0
7	{Presencial,Urgente,Adhesión e-commerce}; sel. factor=3.0
8	{Presencial,Alta,Adhesión e-commerce}; sel. factor=2.0
9	{Telefónico,Baja,Adhesión Tarjeta Crédito}; sel. factor=1.0
10	{Telefónico,Media,Adhesión Tarjeta Crédito}; sel. factor=1.0
11	{Telefónico,Urgente,Adhesión e-commerce}; sel. factor=0.5
12	{e-mail,Baja,Adhesión Tarjeta Débito}; sel. factor=0.3
13	{e-mail,Media,Adhesión Tarjeta Crédito}; sel. factor=0.3
14	{e-mail,Media,Adhesión e-commerce}; sel. factor=0.0
15	{Telefónico,Baja,Adhesión e-commerce}; sel. factor=0.0

FIGURA 5 - EJEMPLO DE TODOS LOS PARES CON CTWEB Y ALGORITMO PROW

El resultado consta de 15 casos de prueba, los cuales cubren todos los pares válidos, teniendo en cuenta los pesos asignados a cada par. Si decidiéramos hacer producto cartesiano con todos los valores de prueba interesantes tendríamos un total de 3 x 4 x 3 = 36 casos de prueba. O sea, **nos estamos ahorrando la ejecución** (y en el caso de pruebas automáticas también la automatización) **de 36 - 15 = 21 casos de prueba que, según la teoría de errores considerada, no agregarían valor a nuestro conjunto de casos de prueba, ya que tienen menos probabilidad 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. No es lo mismo eliminar casos de prueba que tengan combinaciones inválidas, pues tal vez un mismo caso de prueba cubre varios pares, y si eliminamos uno por tener una combinación inválida, podemos estar a su vez eliminando otra combinación que sea interesante a probar y no se pruebe en ningún otro caso de prueba.

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 Figura 6), 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. Si se ingresan valores de clases inválidas tal vez no se logra verificar que para cada entrada incorrecta se hace un manejo adecuado.

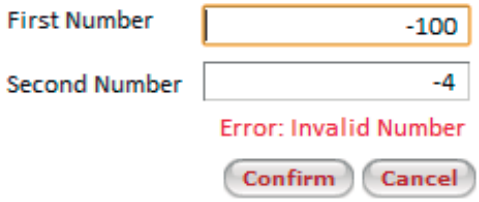
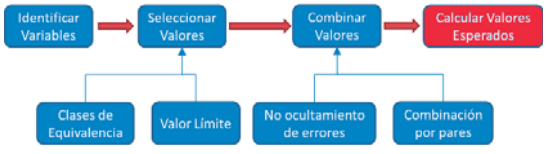


FIGURA 6 - PANTALLA DE EJEMPLO PARA LA TÉCNICA DE NO ENMASCARAMIENTO DE ERRORES

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 a 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. Es importante para que al momento de ejecutar las pruebas no se tengan 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 a 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, que su 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, o 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 podrían incluirse cosas como “se despliega mensaje”, o “se agrega registro a la tabla con los datos ingresados”, “se envía un e-mail”.

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 usar estas técnicas veremos cuándo son más convenientes, en qué tipo de situaciones nos dan más valor. En cambio, las técnicas básicas se podría decir que las utilizamos siempre. Estas las utilizaremos solo para cuando queremos profundizar más, lograr una mejor cobertura, o si la técnica concreta aplica para la situación que estamos 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 técnica de **grafo causa-efecto**. En el caso que la aplicación varíe su comportamiento de acuerdo a distintos estados que se puedan definir en ella, entonces será aplicable la técnica de **máquinas de estado**. Si contamos con casos de uso, podremos aplicar la técnica para derivar casos de prueba a partir de los **casos de uso** (aunque en realidad si no tenemos los casos de uso documentados, podemos comenzar por describir los casos de uso, ya sea entrevistando a los expertos o 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 esta técnica. Para entender cómo funciona luego de describir cómo se modelan los casos de uso, explicaremos cómo se aplica la técnica siguiendo un ejemplo. En ese caso vamos a ver que vamos a tener que tomar muchas decisiones. Vamos a plantear algunas alternativas, o algunas cosas 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. Ganaron su popularidad seguramente con UML y los diagramas de Casos de Uso, como el que aparece en la Figura 7. Ahora, estos diagramas UML simplemente muestran los actores involucrados en cada caso de uso (mostrando 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 se captura conocimiento del negocio y de cómo se desea que trabaje el sistema, describiendo lo que debe hacer sin entrar en detalles. Por este motivo –de estar disponible– es una fuente interesantísima para utilizarla 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.

Lo que ocurre muchas veces es que no se cuenta con las formalizaciones de los casos de uso. Estos quizá están en la cabeza de distintas personas que ocupan distintos roles (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.

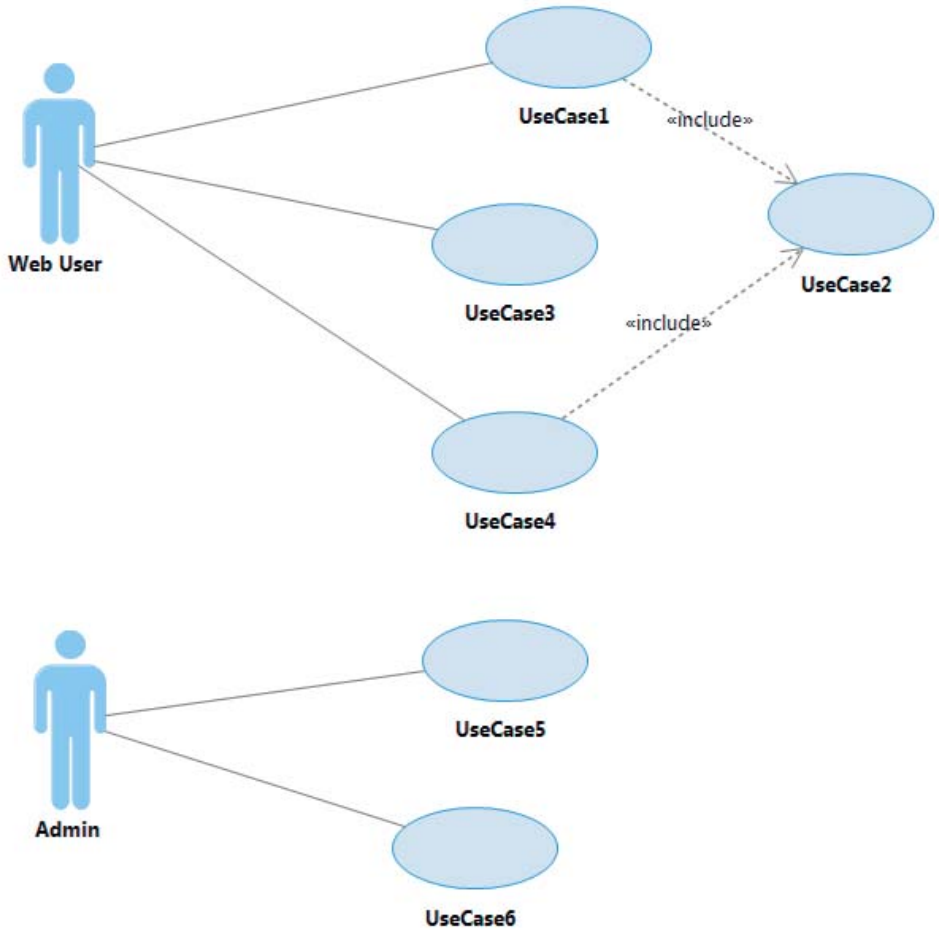


FIGURA 7 - EJEMPLO DE DIAGRAMA UML DE CASOS DE USO

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. Generalmente 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 **pre-condiciones** y **post-condiciones** que deben cumplirse antes y luego de la ejecución del mismo respectivamente. 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. Viendo la Tabla 2, 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, usuario 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 "login" con su usuario y password. Si no cuenta con tal, debe registrarse en el sistema creando así su cuenta.	
Actores Usuario a través de la interfaz web.	
Pre-condiciones El usuario debe estar registrado en el sistema.	
Flujo Normal 1. El usuario accede al sistema en la URL principal. 2. El sistema solicita credenciales. 3. El usuario ingresa proporcionando usuario y password. 4. El sistema valida las credenciales del usuario y le da la bienvenida.	
Flujo Alternativo 1 3. El usuario no recuerda su password por lo que solicita que se le envíe por e-mail. 4. El sistema solicita el e-mail y envía una nueva clave temporal al e-mail. **Flujo Alternativo 2** 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.	
Excepciones E1. Usuario y password incorrectos. Si esto sucede tres veces consecutivas la cuenta del usuario se bloquea por seguridad. E2. (A1). El e-mail proporcionado no está registrado en el sistema. El sistema notifica el error.	
Post-condiciones El usuario accede al sistema y se registra su acceso en la tabla de registro de actividad.	

TABLA 2 - 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, para poder desde ahí visualizar más fácilmente los distintos flujos que se pueden seguir y seleccionarlos 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 saltearse si ven que no les aporta.

La idea 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. Esto muestra que esta técnica en realidad es extrapolable casi que para cualquier especificación del sistema que podamos trasladar a esta representación, 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 Figura 8 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, y al intentar representarlo de este modo, es necesario refinar mucho más en detalles que son muy útiles para su análisis.

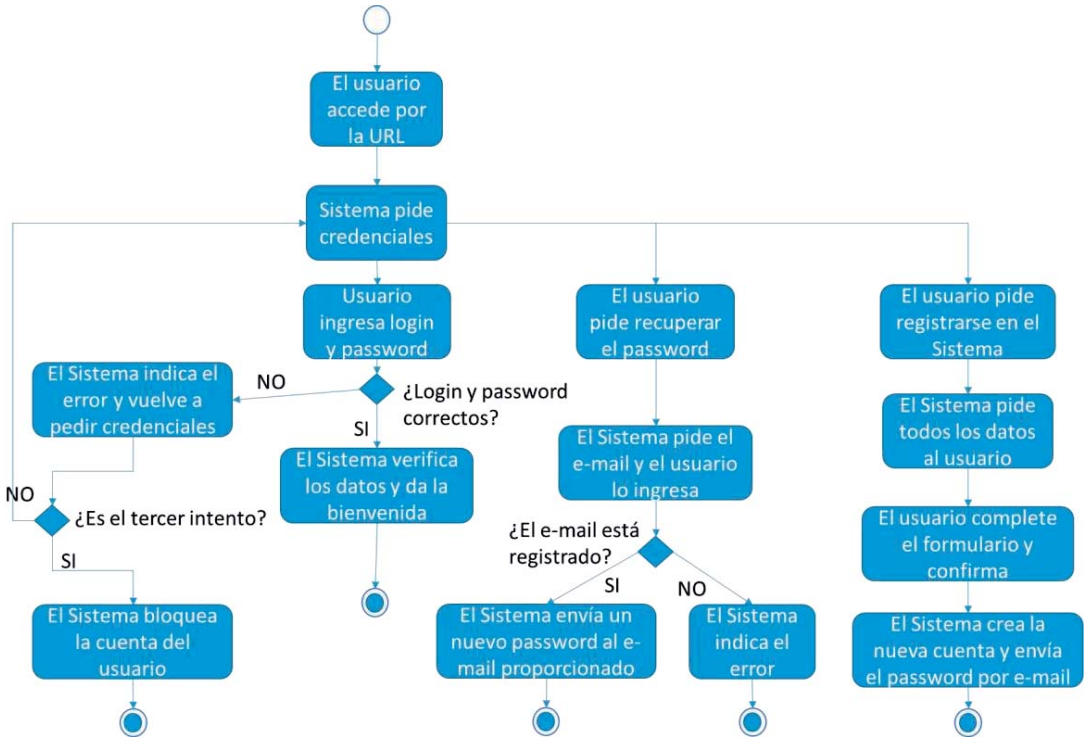


FIGURA 8 - EJEMPLO DE DIAGRAMA DE ACTIVIDAD PARA CASO DE USO DE LOGIN

Por favor testers, ¡abstenerse de buscar errores a la documentación y representación! Incluso, les planteamos como **ejercicio** el analizar las dos representaciones y ver así qué información se agregó al pasar a esta nueva representación. 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 ya aquí surjan oportunidades de mejora, con solo analizar el modelo.

Con una representación de este estilo, y según con el tiempo con el que se cuente para probar, podremos decidir qué flujos queremos probar. Supongo que estamos todos de acuerdo en que esta representación es 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 tenemos 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 de 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 Figura 9 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 rojo están marcados los flujos que se derivarían como casos de prueba.

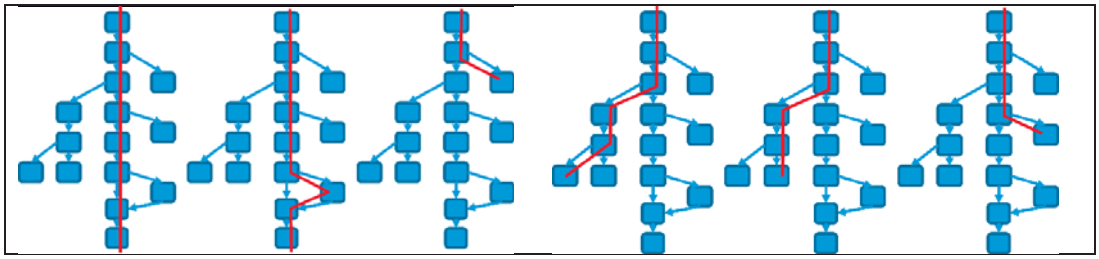


FIGURA 9 - FLUJOS DERIVADOS DEL GRAFO DEL CASO DE USO

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 3 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 Figura 10 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.

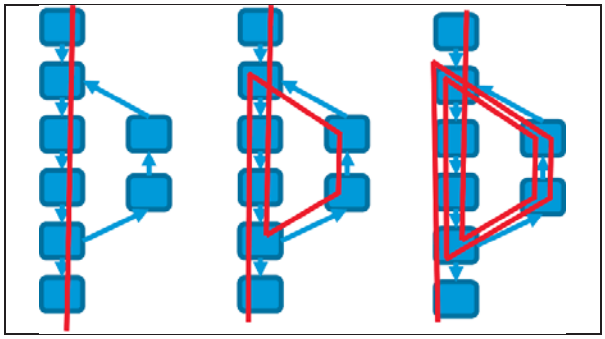


FIGURA 10 - FLUJOS DERIVADOS DEL GRAFO DEL CASO DE USO CON CICLOS

En el ejemplo que veníamos trabajando, existe un bucle para cuando el usuario ingresa el usuario o password 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, esto es necesario hacerlo 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 Tabla 3.

Caso de prueba	Descripción
TC1	Flujo principal. El usuario ya registrado accede con el usuario y password correctos.
TC2	El usuario se equivoca al ingresar usuario y password, pero en su segundo intento lo hace bien y accede al sistema.
TC3	Como el usuario falla en tres intentos ingresando usuario y password, 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 y le asigna un password y se lo envía por e-mail.
TC5	El usuario ya está registrado pero no recuerda su password, por lo que le pide al sistema que le envíe uno nuevo. El sistema genera uno y se lo envía al e-mail.
TC6	El usuario ya está registrado pero no recuerda su password, por lo que le pide al sistema que le envíe uno nuevo. El usuario ingresa una dirección de e-mail que no corresponde a ningún usuario registrado, por lo que termina indicando el error.

TABLA 3 - CASOS DE PRUEBA PARA EL EJEMPLO DEL LOGIN

En la Figura 11 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.

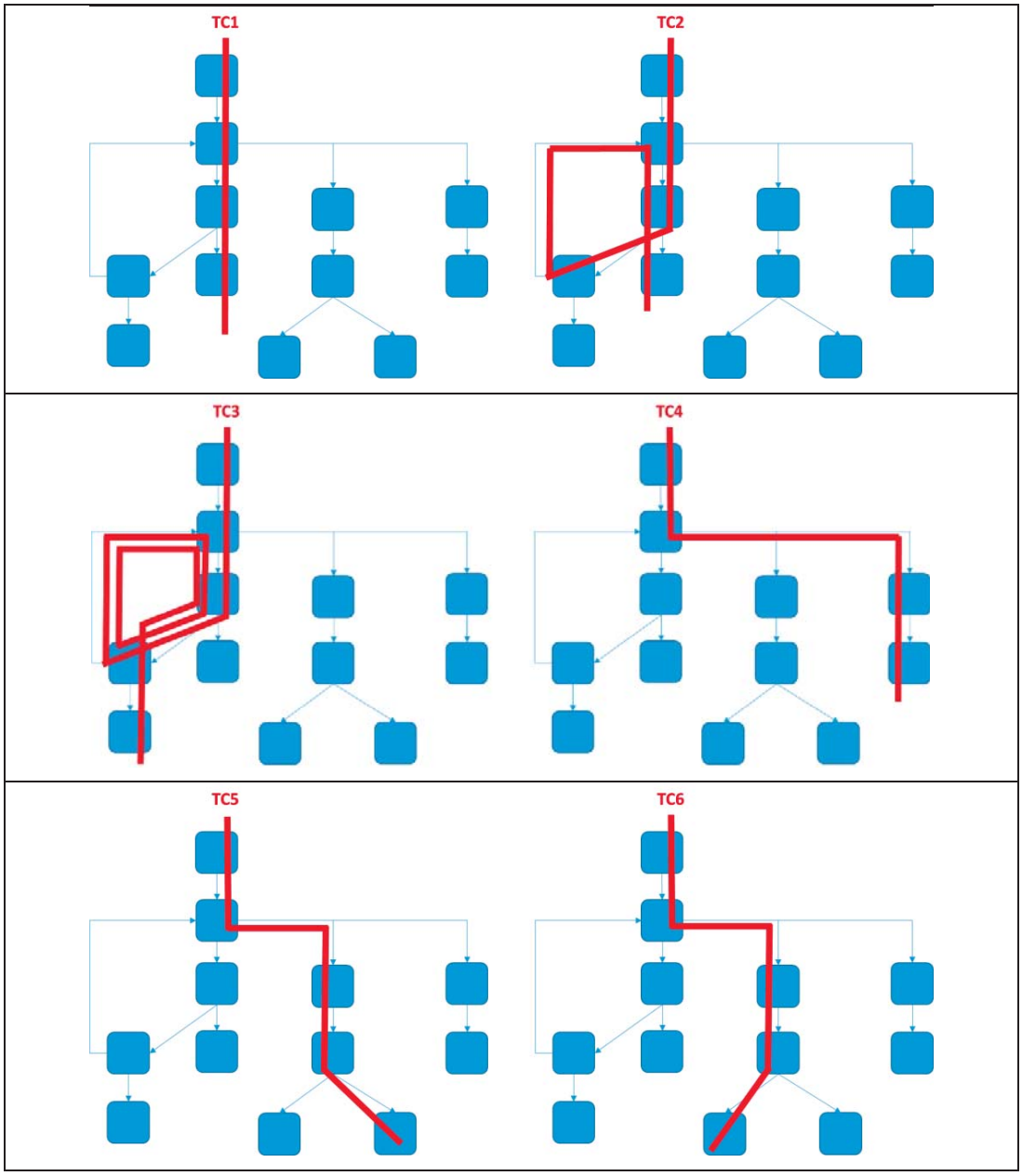


FIGURA 11 - FLUJOS DE LOS CASOS DE PRUEBA DE LOGIN

Observen que como el caso de uso tiene un ciclo, en los casos de prueba generados luego de ejecutar una repetición en T2 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 usuario y password, pero 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 Figura 12.

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.

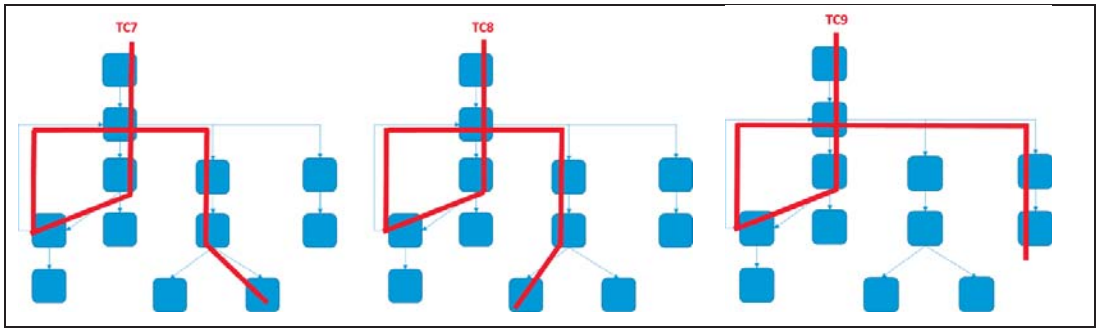


FIGURA 12 - CASOS DE PRUEBA COMBINANDO EL CICLO

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, como 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 el password y para ello se debe ingresar la cuenta de e-mail registrada) solo se puede ejecutar si se selecciona una cuenta de e-mail 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 el usuario y password. El flujo del caso de prueba indica que no son entradas que cambien el comportamiento de ese caso, con lo cual no son variables para este caso,

por lo tanto no tiene sentido diseñar valores de prueba para las variables usuario y password para ese caso en particular.

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

En la Tabla 4 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, pues las variables que tenga cada caso de prueba dependerá su flujo.

Variables => **Caso de prueba**	Usuario y password	E-mail para recordar password	Datos para crear cuenta (nombre, usuario, password, e-mail)
TC1	1 par válido		
TC2	1 par inválido 1 par válido		
TC3	2 pares inválidos 1 par válido		
TC4			1
TC5		1 válido	
TC6		1 inválido	
TC7	1 par inválido	1 válido	
TC8	1 par inválido	1 inválido	
TC9	1 par inválido		1

TABLA 4 - DATOS DE PRUEBA PARA EL EJEMPLO

A su vez, con usuario y password se puede jugar con los valores “par válido” y “par inválido”. Es decir, probar un usuario válido y un password inválido, un usuario inválido (y en ese caso el password no importa), etc. Lo que importa en este caso es si el par es válido o no. En el caso del e-mail, será necesario probar el TC6 con un e-mail inválido, lo cual puede incluir desde e-mails no registrados hasta e-mails 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, y si bien tampoco se indica qué pasa si el usuario ingresa información incorrecta, esto igualmente deberá ser tenido en cuenta también 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, mostrando qué actores están relacionados con cada caso.

Si siempre probamos con el usuario "admin" (volver a ver la Figura 7 en la página 45) 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 al menos algún caso, quizá el flujo principal, sería deseable 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/automatizamos 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:

- Primero seleccionando los casos de uso más importantes
- Para cada caso de uso, cuáles son los flujos más importantes
- Para cada flujo, ver qué datos son los más importantes a utilizar

Cuando digo importante hablo 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 aplicarse cuando existen relaciones lógicas entre variables de entrada o ciertos cálculos involucrando subconjuntos de las variables de entrada. En estas situaciones se apunta a 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 y en esta sección veremos la representación en **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** que represente (obviamente) 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 forma especial de forma que al terminar de completarla ya 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, pudiendo estar en distintos rangos:

- A = IMC < 18,5
- B = 18.5 < IMC < 30
- C = 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 aplicar 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 armando la tabla colocando las condiciones en la columna izquierda. De esta forma comenzaremos diseñando la estructura tal como se ve en Tabla 5. 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							

TABLA 5 - 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 x 3 x 4 = 24 combinaciones posibles. En la Tabla 6 se reflejan las combinaciones interesantes de acuerdo a las acciones que toma el sistema.

Para poder armar la tabla debemos tener en cuenta algo llamado el **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 x 4 = 12 columnas para ese valor (ver la fila "sexo" de la Tabla 6).
- En la variable IMC cada valor está relacionado con la 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 deberían 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								

TABLA 6 - CASOS DE PRUEBA DERIVADOS

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 x 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 la tabla no la armábamos en el orden adecuado la cantidad de casos de prueba resultantes sería distinta, y los casos de prueba resultantes serían redundantes.

Derivación de Pruebas con 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 Figura 13 muestra los constructores básicos para representar estos grafos (considerando los nodos "C" como causas y los nodos "E" como efectos) y los operadores lógicos que se pueden utilizar para representar así las reglas de negocio.

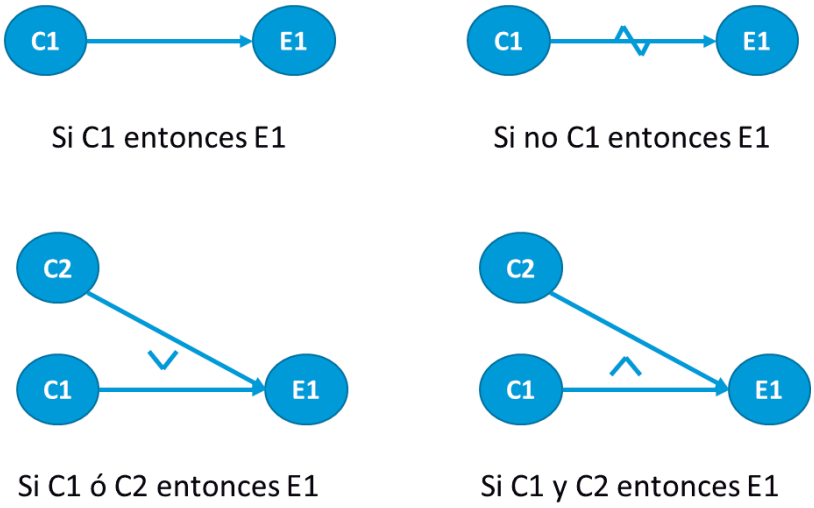


FIGURA 13 - REPRESENTACIÓN CON GRAFOS CAUSA-EFECTO

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

Veamos cómo aplicar esta técnica sobre el mismo ejemplo presentado para las Tablas de Decisión que aplicaba para un sistema que ayuda a determinar el tratamiento a aplicar en base a la edad, sexo y al Índice de Masa Corporal (IMC). La Figura 14 muestra una forma de representar esta lógica con un grafo causa-efecto.

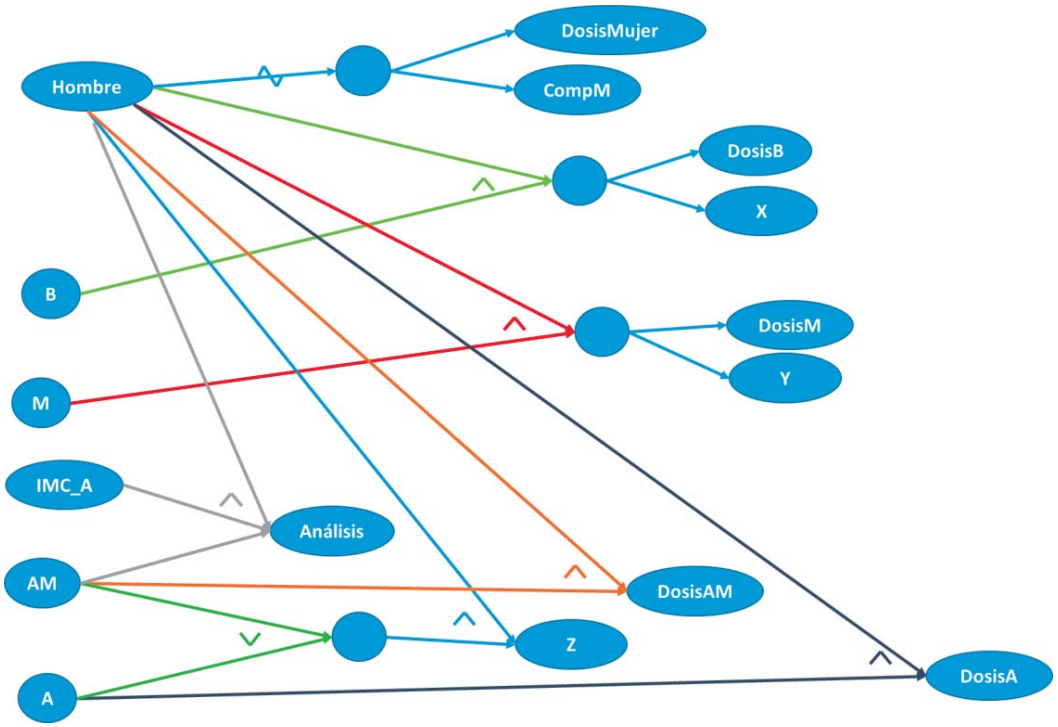


FIGURA 14 - EJEMPLO DE GRAFO CAUSA-EFECTO

Prestar atención 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.[10]

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, entonces la representación puede quedar poco práctica. En esos casos es conveniente comenzar directamente con la representación de Tablas de Decisión, como se mostrará en la siguiente sección.

[10] Puede consultarse esta técnica con mayor detalle en el libro de Myers “The Art of Software Testing”.

Luego, esta representación gráfica se pasa a una tabla que muestre las distintas combinaciones y reglas que se desprenden de lo que ahí está modelado. Para eso se ponen 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, poniendo 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.

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 ver que el sistema indica correctamente cuándo debe hacerse un análisis y cuándo no, podemos generar pruebas pensando específicamente en esa situación, viendo qué combinación de causas deben 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 (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).

Por ejemplo, en la Tabla 7 se muestra la tabla derivada a partir del grafo de la Figura 14. 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 7 - TABLA DE DECISIÓN PARA EL GRAFO CAUSA-EFECTO

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

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

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 por ejemplo 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 pueden ver parte del trabajo de Harry Robinson en esta página:

- http://www.geocities.com/model_based_testing/.

Aplicación de la Técnica en un Ejemplo

Veamos un caso de prueba de ejemplo para la Gestión de Artículos, que quizá sea el concepto central en cualquier sistema de *Retail* de la operativa de cualquier empresa.

Un posible enfoque para el diseño de casos de prueba es 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 Figura 15 presenta una máquina de estados parcial para el concepto *artículo*.

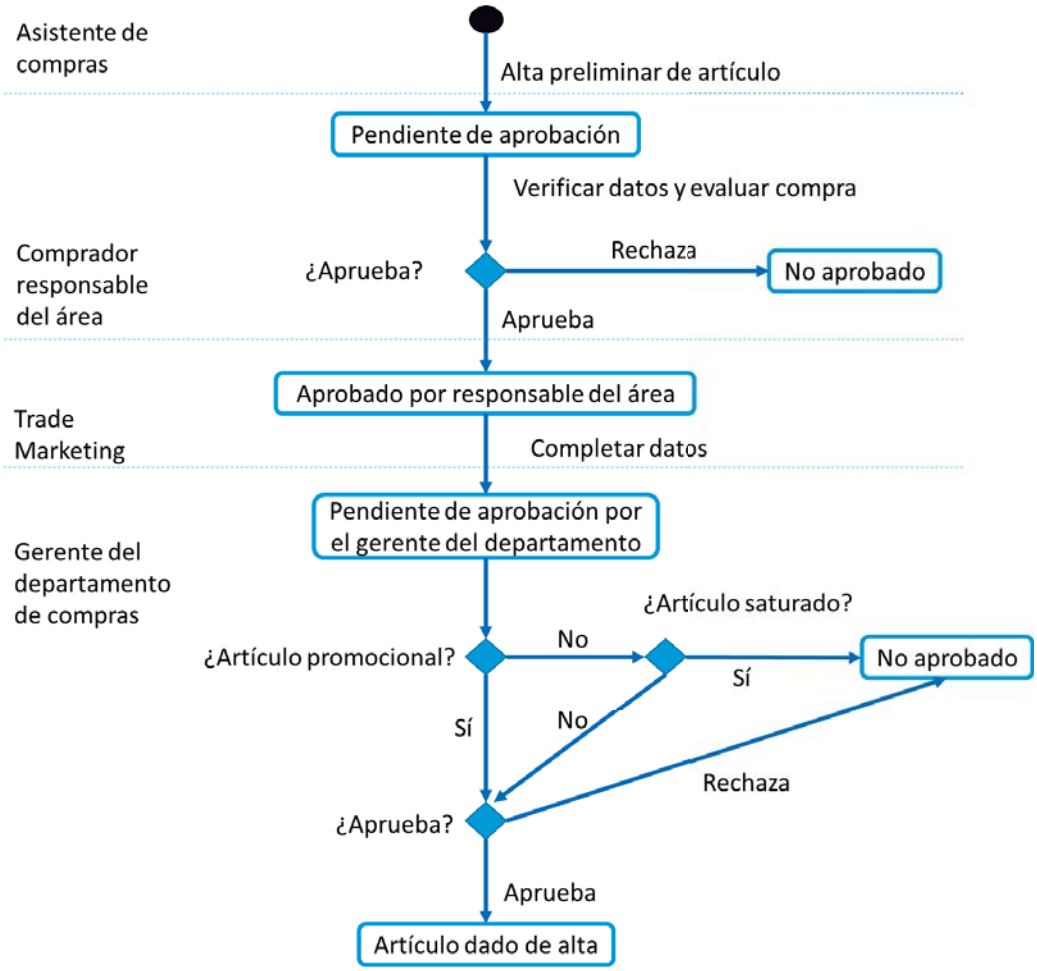


FIGURA 15 - MÁQUINA DE ESTADOS PARA SISTEMA DE GESTIÓN DE ARTÍCULOS

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 siguiendo un algoritmo definido. El resultado de aplicar ese algoritmo 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.

CTweb para Derivar Pruebas con Máquinas de Estado

Si bien esta técnica puede usarse en forma manual, aprovecharemos la herramienta CTweb (ctweb.abstracta.com.uy) que nos permite diseñar los casos de prueba que cubren una máquina de estados dada, ya sea en formato texto como en UML (recordar que ya hablamos de esta herramienta en el capítulo anterior ya que es útil también para combinar datos de prueba).

CTweb permite especificar la máquina de estados y el criterio de cobertura que queremos alcanzar, y devuelve los recorridos sobre esa máquina para satisfacer el criterio, o sea, los casos de prueba.

Las máquinas de estado pueden diseñarse con alguna herramienta de modelado UML estándar, tal como Papyrus para Eclipse (http://www.eclipse.org/papyrus/), o se representan fácilmente en un archivo de texto.

Pensemos en otro ejemplo, un caso tan simple como muestra la Figura 16 para gestionar solicitudes.

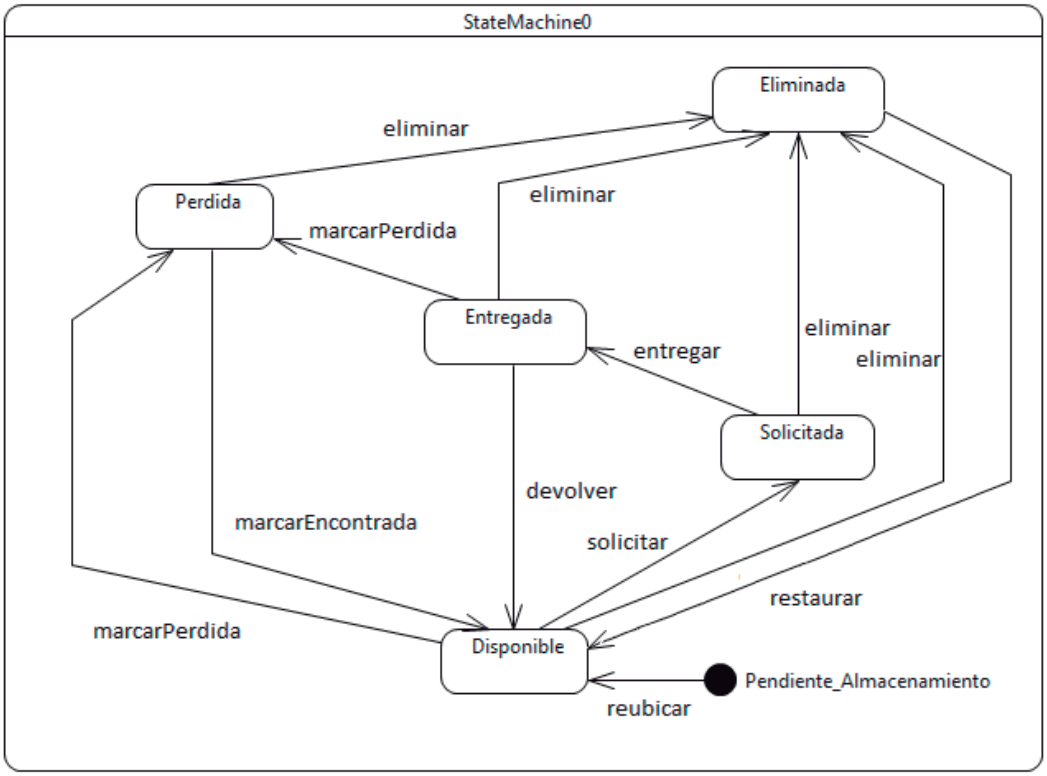
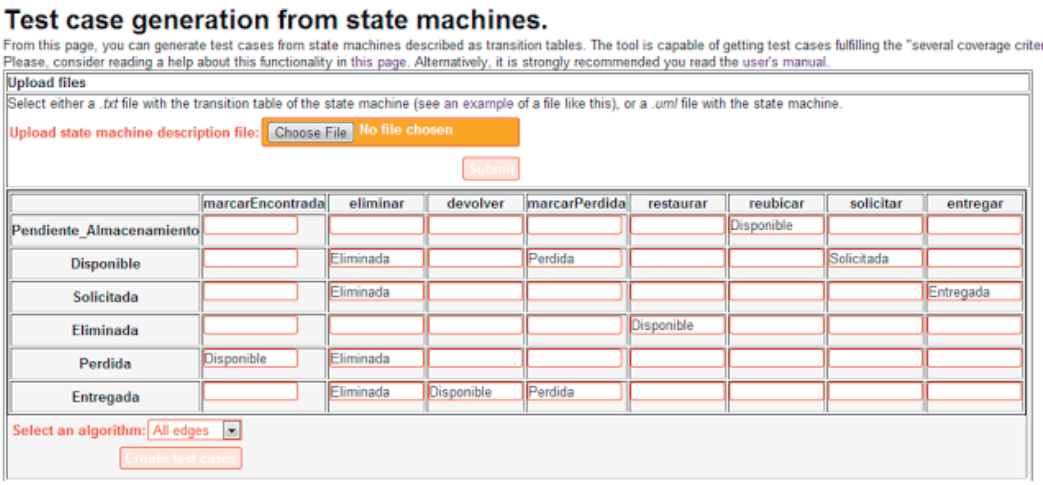


FIGURA 16 - MÁQUINA DE ESTADOS PARA GESTIONAR SOLICITUDES

Esta máquina de estados está modelada con Papyrus, y al guardarla se obtiene un archivo UML en formato XMI estándar que CTweb puede procesar.

Luego de cargarlo en CTweb se ve como muestra la Figura 17.



Test case generation from state machines.

From this page, you can generate test cases from state machines described as transition tables. The tool is capable of getting test cases fulfilling the "several coverage criteri
Please, consider reading a help about this functionality in this page. Alternatively, it is strongly recommended you read the user's manual.

Upload files

Select either a *.txt* file with the transition table of the state machine (see an example of a file like this), or a *.uml* file with the state machine.

Upload state machine description file: Choose File No file chosen

Submit

	marcarEncontrada	eliminar	devolver	marcarPerdida	restaurar	reubicar	solicitar	entregar
Pendiente_Almacenamiento						Disponible		
Disponible		Eliminada		Perdida			Solicitada	
Solicitada		Eliminada						Entregada
Eliminada					Disponible			
Perdida	Disponible	Eliminada						
Entregada		Eliminada	Disponible	Perdida				

Select an algorithm: All edges

Create test cases

FIGURA 17 - MÁQUINA DE ESTADOS CARGADA EN CTWEB

Esto nos da distintos resultados, según la cobertura que queremos obtener sobre el modelo. Por ejemplo, para el criterio de transiciones el resultado es:

1) [reubicar, solicitar, entregar, devolver]
2) [reubicar, solicitar, entregar, marcarPerdida, marcarEncontrada]
3) [reubicar, solicitar, entregar, marcarPerdida, eliminar, restaurar]

Para el de all-pairs obtenemos:

1) [reubicar, solicitar, entregar, devolver, solicitar, eliminar, restaurar, solicitar]
2) [reubicar, eliminar, restaurar, eliminar]
3) [reubicar, marcarPerdida, marcarEncontrada, solicitar]
4) [reubicar, solicitar, entregar, devolver, eliminar]
5) [reubicar, solicitar, entregar, devolver, marcarPerdida, eliminar, restaurar, marcarPerdida]
6) [reubicar, marcarPerdida, marcarEncontrada, eliminar]
7) [reubicar, marcarPerdida, marcarEncontrada, marcarPerdida]
8) [reubicar, solicitar, entregar, eliminar, restaurar]
9) [reubicar, solicitar, entregar, marcarPerdida, marcarEncontrada]
10) [reubicar, solicitar, entregar, marcarPerdida, eliminar]
11) [reubicar, solicitar, entregar, marcarPerdida]
12) [reubicar, solicitar, entregar, eliminar]

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.

MATRIZ CRUD

Cada entidad de un sistema de información tiene un ciclo de vida: todo nace, crece (se actualiza) y 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.

Un ejemplo de este análisis para AjaxSample[11] se puede ver en la Tabla 8. Por ejemplo, en "New Invoice" se puede ver que hay una lectura y actualización en la entidad de Client, pues se leen los datos del cliente para poder seleccionar a quién corresponde la factura, y se actualiza el balance. Lo mismo con producto, 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 (pueden existir datos maestros que no se puedan crear, sino que se dan de alta por un sistema externo, o directamente por un administrador de la base de datos, o hay datos que no se deben eliminar nunca, etc.).

[11] AjaxSample http://samples.genexus.com/ajaxsample/home.aspx

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

TABLA 8 - MATRIZ CRUD PARA AJAXSAMPLE

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 de que cada vez que un dato se actualiza en algún lugar se refleja en cada pantalla donde este es consultado.

Comentarios Finales del Capítulo

¿Con esto ya estamos listos para probar lo que sea? Bueno, tenemos una base, pero siempre es importante investigar en 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.

¿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. Les recomendamos en particular los materiales proporcionados por Myers, Bach, Bolton, Weinberg, etc.
- 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 (contacto@abstracta.com.uy).

“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 de 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 el lector luego pueda profundizar con un panorama de lo que ésta incluye.

¿Qué es el Testing Exploratorio?

Veamos algunas definiciones provistas por algunos de los gurúes en el tema. En el año 1983 Cem Kaner le ponía un nombre a esta metodología de trabajo, y su definición[12] es la siguiente:

> *"Es un estilo de trabajar con 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 tales 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[13] más populares, como las siguientes:

> *"El Testing Exploratorio consiste en el aprendizaje, diseño y ejecución de pruebas de forma simultánea."*

Esa misma definición se amplía para proveer un poco más de detalles de la siguiente manera:

> *"El Testing Exploratorio es cualquier testing que se realiza hasta el punto en el que el tester controla activamente el diseño de las pruebas, mientras que esas pruebas se ejecutan y posteriormente se hace uso de la información obtenida mientras se prueba, para diseñar nuevas y mejores pruebas."*

A partir de las definiciones, y tal como lo veremos a lo largo de este capítulo, el testing exploratorio presenta una estructura externa que es sencilla de describir, en la que durante un lapso de tiempo un tester –o equipo de testers– interactúa con un producto para cumplir con el objetivo de una misión, para posteriormente presentar y reportar los resultados que el resto de los actores del proyecto utilizarán para tomar decisiones a conciencia. 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

---

[12] "Exploratory Testing After 23 Years", Cem Kaner, 2006

[13] "Exploratory Testing Explained", James Bach, 2003

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 mencionábamos anteriormente, 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 contra un sistema, o 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"[14], 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, por lo que comienza a explorar, tomar apuntes de sus hallazgos y conclusiones, mejorando el aspecto del mapa. Un explorador debe de ir alerta a lo que pueda surgir, debe caminar teniendo su mente abierta y receptiva.

[14] "The Permanent Book of Exploration", John Keay

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 más precisamente 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 un explorador, un tester 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 utilizarlas 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.

Propiedades del Testing Exploratorio

Ahora que hemos visto algunas definiciones, veamos algunas propiedades que además, describen dónde nos puede aportar valor como práctica:

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 prueba.
4. Es un acercamiento a las actividades de testing que consisten 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, extraídos del libro "*Exploratory Software Testing"* de James Whittaker. 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, responsabilidad, y 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" como:

1. Para referirse a lo que se dice o hace solo para 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, o bien para tener un primer contacto con la aplicación antes de pasar a una exploración con técnicas más sofisticadas.

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 contra 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 el éxito de la misma 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 de un tester sucede que en determinado momento al enfrentar un nuevo sistema uno diseña pruebas basándose en alguna estrategia existente, pero lo hace quizá en forma inconsciente. Quizá pensamos 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 queremos ejecutar cada uno de los estados, aplicando así inconscientemente una técnica basada en máquinas de estado.

Claramente, 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 seguramente 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 administración del testing exploratorio. 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 van desde los 45, 90 y 120 minutos de duración. Esto es principalmente con el propósito de fijar la atención en la prueba en forma sostenida, y que ningún aspecto nos pase desapercibido, 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 pueden identificarse riesgos que no fueron considerados originalmente y desestimarse 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 para su planificación, como para la actualización de su planificación. También, recordemos que como lo definiera Cem Kaner, éste acercamiento al testing enfatiza la responsabilidad y la libertad del tester, 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, contar con un mecanismo como las sesiones, que brinde estructura y organización, es necesario. 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 estado hablando de las sesiones, de algunas de sus características, como 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: ¿Voy 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. La misma está pensada con una orientación a encontrar problemas relacionados con la seguridad de una aplicación:

> *"Realizar ataques por inyección SQL 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 deber ser capaz de proveer de dirección a quien la lea sin restringir las acciones o tareas de testing a realizarse. 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 revisen la misma.

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.

> *"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 aun así, no estar seguros 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 ***W.B.S.*** 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 unas 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.

Algo que debemos tener en cuenta, es que una buena misión no solo nos sirve de inspiración al momento de llevar adelante nuestras actividades de testing, sin por ello darnos órdenes, o acciones precisas de lo que debemos de hacer o encontrar, sino que también contienen información que provee valor al resto de los actores del proyecto, que de no ser así diríamos que no estamos usando adecuadamente nuestro tiempo.

Anatomía de una Sesión

Anteriormente, dijimos que una sesión debía contar con tres grandes características, como 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 Figura 18).

FIGURA 18 - 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 un tester se topa con un comportamiento que podría significar un problema.
3. **Armado de la Sesión:** Son todas aquellas tareas de las que participa un tester, 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 de 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 apunta a 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 sub-sección 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 prueba.

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 apunta a invertir una mayor proporción del tiempo en probar el sistema bajo prueba, 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 prueba, 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 prueba, archivos de *mockups*, de especificaciones, consultas SQL, capturas de pantalla, entre otros. Algo a tener en cuenta, es que los archivos que aquí se detallen, 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 ellas 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***. Además, al término de una sesión las notas serán utilizadas para la construcción de nuevas pruebas, como también, para brindar información al resto del equipo sobre ***qué*** testeamos, ***dónde*** enfocamos nuestros esfuerzos y ***porqué*** creemos que nuestras pruebas podrían ser consideradas como *suficientemente buenas*.

8. Defectos

Tal como su nombre lo adelanta, aquí vamos a registrar tanto los ID's 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, éste último tiene por cometido el registrar todo inconveniente o impedimento que haya surgido quitando tiempo a la

sesión y la concreción de su objetivo. Habitualmente, los problemas o inconvenientes que se registran, pueden consistir en potenciales defectos, problemas de ambiente, 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 prueba de ejemplo.

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 REPORTES 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 ambiente 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 tester o 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:
    - El ***Armado de la Sesión***;
    - ***Testing***; y,
    - La ***Investigación y Reporte de Defectos***.

En la Tabla 9 vemos un ejemplo de un reporte clásico, conteniendo dos sesiones clásicas basadas en el modelo creado por James y Jonathan Bach[15] el cual puede generarse a partir del uso de una herramienta gratuita disponible bajo el nombre ***CLR-Sessions[16].***

[15] "Completed Sessions Reports", James Bach, http://www.satisfice.com/sbtm/demo/reports/s_by_ses.htm

[16] Acceso a CLR-Sessions: https://github.com/aquaman/clr-sessions/archive/master.zip

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

TABLA 9 - 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 la misma.
- **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 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 éste 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 trabajo dedicado a la búsqueda de defectos.
- **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 ésta categoría se incluye el buscar información, armar la sesión como tal, configurar un ambiente, 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 ambientes de prueba, problemas con la infraestructura edilicia que afecte los ambientes, 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;
- Para 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 ('computer-assisted testing').”

– 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 Duft Punk) *harder*, *faster*, *stronger*… caos. En este capítulo la idea es hablar de automatización de pruebas funcionales en general, mostrando 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 tiene la idea de convencer de que la automatización es la varita mágica que va a lograr que nuestras pruebas sean mucho mejores.

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 opción más popular y la más usada.

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. Tienen el objetivo de verificar que el producto no ha sufrido regresiones.

¿Por qué se llaman pruebas de regresión?

En un principio pensaba que se refería a *regresar* a ejecutar las mismas pruebas, ya que se trata un poco de eso. Luego vi que el concepto en realidad está asociado a verificar que lo que estoy probando ***no tenga regresiones***. El tema es que "no tener regresiones" imaginé que se refería a que no haya una regresión en su calidad, o en su funcionalidad, pero escuché el rumor de que el concepto viene de la siguiente situación: si los usuarios tienen la versión N instalada, y le instalamos la N+1, y esta tiene fallos, nos veremos atormentados al tener que volver a la versión previa, hacer una ***regresión*** a la versión N. ¡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 andaba 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 previamente diseñadas.

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. James Bach y Michael Bolton, dos de

los *expertos* en testing, comentan en un par de artículos[17] las diferencias entre el testing y el chequeo. El testing es algo donde uno pone 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. 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 uno desea que algo funcione e inconscientemente confunde lo que ve para dar un resultado positivo.

¡Ojo! ¡No estamos diciendo que el testing sea aburrido! Al menos 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 en cómo podría programar algo para no tener que hacer esta tarea a mano. Ahí es donde podemos introducir testing automatizado ya que los robots no se aburren.

El **testing automatizado consiste en** que una máquina logre ejecutar los casos de prueba en forma automática, leyendo la especificación del mismo de alguna forma, que pueden ser scripts en un lenguaje genérico o propio de una herramienta, a partir de planillas de cálculo, modelos, etc. Por ejemplo Selenium[18] (de las más populares herramientas open source para automatización de pruebas de aplicaciones Web) tiene un lenguaje llamado Selence, brindando 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 JUnit[19] en Java, y a otros entornos de ejecuciones de pruebas.

---

[17] Referencia a los artículos: 1) http://www.developsense.com/blog/2009/08/testing-vs-checking/ 2)http://www.satisfice.com/blog/archives/856

[18] Selenium: herramienta de pruebas web automatizadas. http://seleniumhq.org

[19] JUnit: herramienta de pruebas unitarias automatizadas. http://junit.org

A continuación compartimos algunas ideas tomadas de distintas charlas con varias personas con 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, dándome lugar a dedicar mi tiempo a otras pruebas, con lo cual puedo garantizar más calidad del producto.*

Desarrollador:

> *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 surgen otro tipo de cuestiones como ¿Las pruebas de regresión hay que automatizarlas 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

The cost of a single defect in most organizations can offset the price of one or more tool licenses (extraído de “Surviving the Top 10 Challenges of Software Test Automation” de Randall W. Rice).

¿Costo o inversión? Generalmente se dice que el testing automatizado permite ampliar la cobertura y alcance de las pruebas, reducir costos, poner el foco de 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.

Es interesante apuntar lo que se plantea en un artículo[20] de Paul Grossman, publicado por un importante proveedor de la industria del software y hardware, donde se habla de los beneficios de la automatización de pruebas en términos financieros, más que nada para dar más argumentos a quien quiere visualizar o mostrar el valor que tiene este tipo de inversiones (ya que verán al final que no se trata de un gasto, sino de una inversión para ganar no solo en calidad sino en ahorro de dinero).

En cierta forma pondremos esos beneficios en duda.

Para saber si "invertir" analicemos si esa inversión es un costo que tiene la calidad, o si también significará ahorros en otros costos asociados a la falta de calidad. Para esto tenemos que verlo en números para entender y creer.

Para las pruebas, aunque sean "automáticas" y "ejecutadas por una máquina", vamos a necesitar gente capacitada. Como bien dice Cem Kaner en un capítulo[21] dedicado a automatización, habla de que una herramienta no enseña a tus testers cómo testear, si el testing es confuso las herramientas refuerzan esa confusión. Recomiendan 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.

---

[20] Referencia al artículo: http://h20195.www2.hp.com/v2/GetPDF.aspx/4AA2-4175EEW.pdf

[21] Extraído de *Lessons Learned in Software Testing (Cem Kaner, James Bach, Bret Pettichord) Lesson 106 – A test tool is not a strategy*)

Veamos con un ejemplo hipotético que sea más entendible por alguien que decide sobre las finanzas de una empresa:

Un tester 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 plantea que un tester en promedio cuesta 50u$s la hora (¡ojalá en Uruguay fuera igual!) y si es senior cuesta 75u$s (¡divino!). Siendo así serían 400u$s o 600u$s respectivamente por día, pero en el segundo caso serían 16 horas más que en el primero (esto es porque considera que el senior es el que sabe automatizar, el otro solo se considera para test manual). Además los test automatizados corren más rápido que los test manuales, en promedio digamos que 5 veces más rápido (generalmente es más, pero no exageremos). Si hablamos de un equipo de 10 personas, 5 senior y 5 para test manual, tenemos un costo mensual de 105.000u$s (168hs por mes, o sea 8 horas por 21 días, 42.000 en test manual y 63.000 de testers senior). Esto en principio nos da un total de 1.350 horas a un costo de 78u$s la hora (acá se está considerando que de las 168 horas mensuales de trabajo, se estará ejecutando 135, lo cual podría suponerse como razonable si se considera cualquier *overhead* como enfermedad, vacaciones, capacitación, etc.). Si ponemos a 3 de los senior a automatizar, el costo seguirá siendo el mismo pero tendríamos 16hs x 3 seniors x 21 días x 5 veces más test por hora, lo que nos da un agregado de 5.040 horas mensuales equivalentes a horas de test manual. Si consideramos el resto del equipo haciendo pruebas manuales (7 x 135) son 945 horas más, lo que da en total 5.985hs de test con un costo de 17,5u$s la hora (105.000u$s / 5.985 horas). Esto es un beneficio expresado en términos financieros, que se entiende fácilmente. Pasamos de pagar la hora de test de 78u$s a 17,5u$s, o visto de otro modo, aumentamos las horas de prueba de 1.350 a 5.985 bajo el mismo costo. Se muestra el resumen de este razonamiento en la Tabla 10.

Manual	Automatizado
Horas 10 x 135 = 1.350 horas	Horas 3 x 21 x 16 x 5 = 5.040 horas 7 x 135 = 945 Total: 5.985 horas
Costos 78u$s por hora	Costos 17,5u$s por hora

TABLA 10 - COMPARACIÓN DE COSTOS POR HORA

Claramente esto 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 de Eduardo Galeano).

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 más cuentas para visualizarlo mejor).

Es conocida en ingeniería de software la gráfica que muestra el costo de corregir un defecto detectado según en la etapa que se haya encontrado (desarrollo, integración, beta testing, producción). De hecho, he leído en un foro que querían hacer una apuesta sobre cuántas charlas (en una determinada conferencia) iban a comenzar mostrando esa gráfica. ¡Si será popular!

En la Tabla 11 se muestra un estudio de cuántas horas en promedio cuesta corregir un defecto según la etapa, y considerando costo por hora, cuánto cuesta en dólares. Esto no está considerando siquiera los costos ocultos que hay como la pérdida de imagen, confianza, e incluso el desgaste del equipo.

Defecto introducido en codificación	Etapa en la que se detecta			
	Codificación/ pruebas unitarias	Integración	Beta test	En producción
Corrección	3,2 hs	9,7 hs	12,2 hs	14,8 hs
Costo fijo	240 u$s	728 u$s	915 u$s	1.110 u$s

TABLA 11 - COSTO DE CORREGIR ERRORES SEGÚN ETAPA

Esta tabla está hecha en base a un estudio hecho con empresas de desarrollo de software de intercambio financiero electrónico[22]. Muestra el costo en horas de reparar un bug según la fase en la que se encuentra. Luego se aplicó un costo por hora de 75u$s.

Lo que se desprende de esto es algo ya muy conocido: cuanto antes se encuentren los bugs, mejor. Y como vamos a tener testers que trabajan 24hs al día, ejecutando más rápido, es más probable que se encuentren más errores antes de pasar las versiones a beta testing o a producción. Es difícil estimar cuántos, pero por cada bug que se encuentre más temprano se

[22] Planning Report 02-3, "The Economic Impacts of Inadequate Infrastructure for Software Testing", Prepared by RTI for National Institute of Standards and Technology, May 2002, p 7-12.

ahorrará al menos 200u$s…. ¡nada mal! Esto se puede ajustar al precio que cuesta la hora de los desarrolladores en su equipo y seguirá siendo rentable.

En estos números se llega a la conclusión de cuánto cuesta un defecto encontrado en cada una de las distintas etapas.

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, evitando problemas de operativa, pérdida de imagen de clientes e incluso evitando problemas legales.
- *IT value*: Mejora el trabajo del grupo de IT ya que le simplifica las tareas rutinarias, permitiendo que con el mismo costo hagan mucho más y mejor.

¿Cuándo se Hacen Visibles los Resultados?

Tal vez uno piense que si las pruebas me encuentran un error es ahí cuando estoy encontrando beneficios, entonces voy a poder 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 me está diciendo que las pruebas que ya automaticé están cumpliendo lo que verifican. Un artículo de *Methods and Tools*[23] dice que se encuentran gran cantidad de bugs al momento de automatizar los test cases. Al automatizar uno debe explorar la funcionalidad, probar con distintos datos, etc. Generalmente lleva un tiempo en el que uno 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 automatizo pruebas sobre un módulo y considero que con esas pruebas es suficiente, entonces ¿no lo pruebo más? el riesgo es que las pruebas automatizadas no estén cubriendo todas las funcionalidades (paradoja del pesticida). Dependencia con la

[23] Referencia al artículo: http://www.methodsandtools.com/archive/archive.php?id=33

calidad de los test. Tal vez tengo mil pruebas, y por eso creo que tengo 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[24] (que complementa la información que nos aporta un tester que ejecuta 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 permita automatizar procesos manuales, trayendo 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 mejoran 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 bien esta es una idea que la venimos afirmando desde el comienzo del capítulo, queremos plantear una teoría a la que le llamamos "acumulatividad nula"[25]. Para eso veamos la gráfica en la Figura 19, donde se muestra que las *features* crecen cada vez más a lo largo del tiempo (de una versión a otra), pero *el test* no (no conozco de 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 ir creciendo también proporcionalmente. De aquí sale el problema de no tener tiempo para automatizar, pues no hay siquiera tiempo para el test manual.

---

[24] Extraído de "Lessons Learned in Software Testing" de Cem Kaner, James Bach, Bret Pettichord

[25] 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.

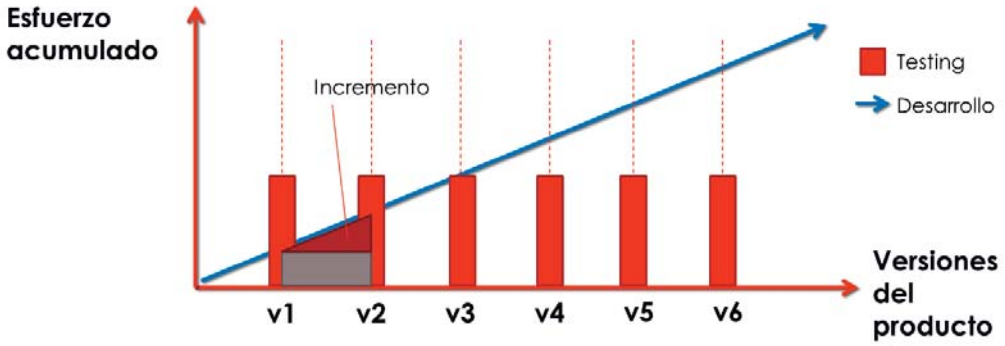


FIGURA 19 - REPRESENTACIÓN DE LA ACUMULATIVIDAD NULA

Un colega argentino llamado Ernesto Kiszcurno[26], de una consultora dedicada a calidad y procesos de ingeniería, dice 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 cambian 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 pasa el tiempo y crece 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 de por qué automatizar muy importante (y muy mencionado) 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

[26] http://www.ernestokiszkurno.com.ar/

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 dejan 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**.

Al 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**, pudiendo 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 ejecute sobre un entorno u otro.

¿Automatizar o no Automatizar?

Como ya dijimos, luego de diseñar las pruebas, hay que ejecutarlas cada vez que hay 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 generalmente es necesario conocer un poco más a fondo la forma en que trabaja la herramienta, manejar juegos de datos de

pruebas, administrar los ambientes 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 podrá volver a testear lo que ya se automatizó con bajo esfuerzo, permitiendo que el equipo de pruebas aumente el tamaño del conjunto de pruebas, aumentando 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 Figura 20 muestra uno de los factores más importantes, y es la cantidad de veces que vamos a repetir la ejecución de un caso de prueba.

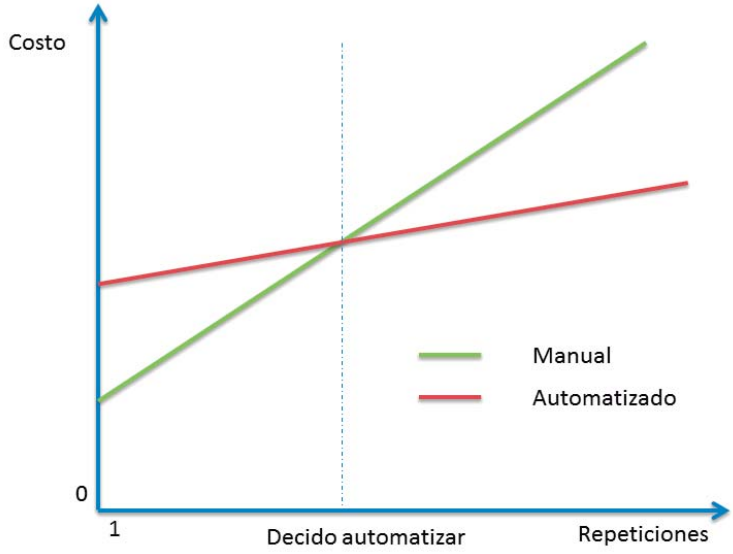


FIGURA 20 - DECISIÓN DE AUTOMATIZAR O NO AUTOMATIZAR

El costo de una sola repetición es mayor para el automatizado, pero las pendientes son distintas. Hay un punto en el que las rectas se cruzan, y ese es el número de repeticiones

que marca la inflexión, si el caso de prueba lo ejecutaremos 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, o 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:

- *Automating chaos just gives faster chaos*. (*James Bach*)
- *If you do not know which tests are the most important and which tests are the most applicable for automation, the tool will only help perform a bad test faster*. (Extraído de "*Software test automation: effective use of test execution tools*" de *Mark Fewster & Dorothy Graham)*.

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 viendo 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.

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).

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, como es el caso del lenguaje Selense de la herramienta Selenium, o pueden ser una biblioteca o API para un lenguaje de propósito general como lo es JUnit para Java.

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. Otras trabajan a nivel de protocolos de comunicación, entonces tendremos acciones relacionadas con los mismos, como puede ser a nivel de http la herramienta HttpUnit[27], la cual nos da la posibilidad de ejecutar GET y POST a nivel de protocolo.

En la Figura 21 se puede apreciar que en una prueba JUnit se invoca una funcionalidad del sistema directamente sobre un objeto bajo pruebas. Se utiliza un valor de entrada para sus parámetros y se verifican los parámetros de salida. En este caso la ejecución es sobre un objeto, en el caso de Selenium los parámetros se cargarán sobre inputs existentes en las páginas web, y luego la ejecución de la funcionalidad será haciendo submit sobre el botón correspondiente. En la Figura 22 se observa un ejemplo de una prueba automatizada con Selenium. 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.

```
public class TestSuite {
  @Test
  public void testCase01() {
    String testData = "input";
    String expected = "expectedOutput";
    String actual = systemUnderTest.execute(testData)
    assertEquals("failure", expected, actual);
  }
```

FIGURA 21 - EJEMPLO DE PRUEBA CON JUNIT

Para preparar pruebas automatizadas siguiendo este enfoque es necesario programar los scripts. Para esto es necesario conocer el lenguaje o API 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.

[27] HttpUnit: http://httpunit.sourceforge.net/

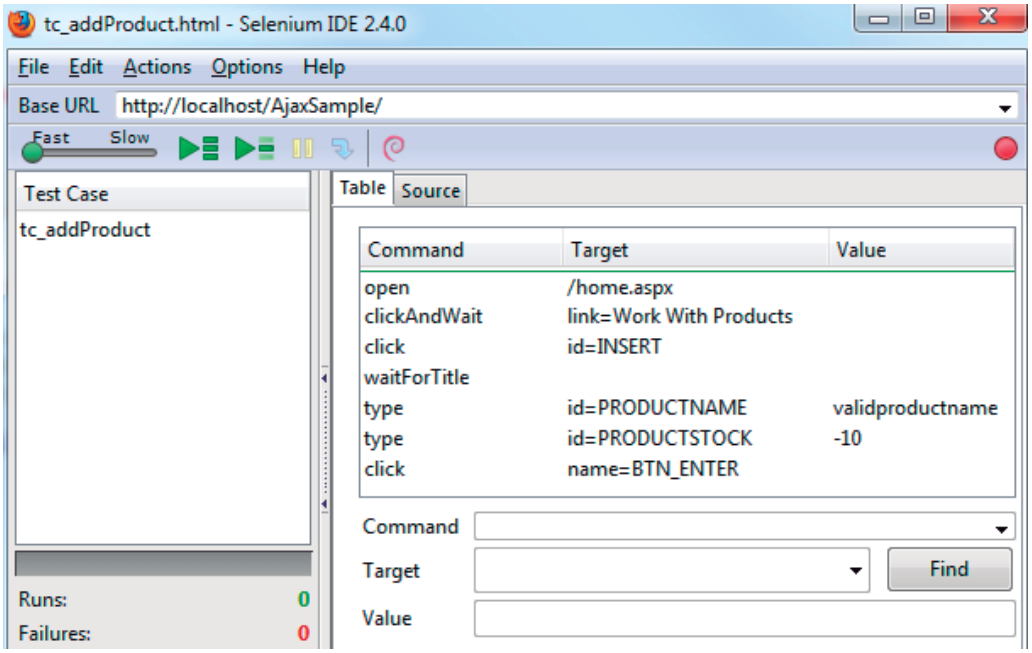


FIGURA 22 - EJEMPLO DE PRUEBA CON SELENIUM

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 Figura 23 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.

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. Tal vez sería realizable por un desarrollador, pero para un tester puede ser más complejo, y por eso es deseable contar con esta funcionalidad.

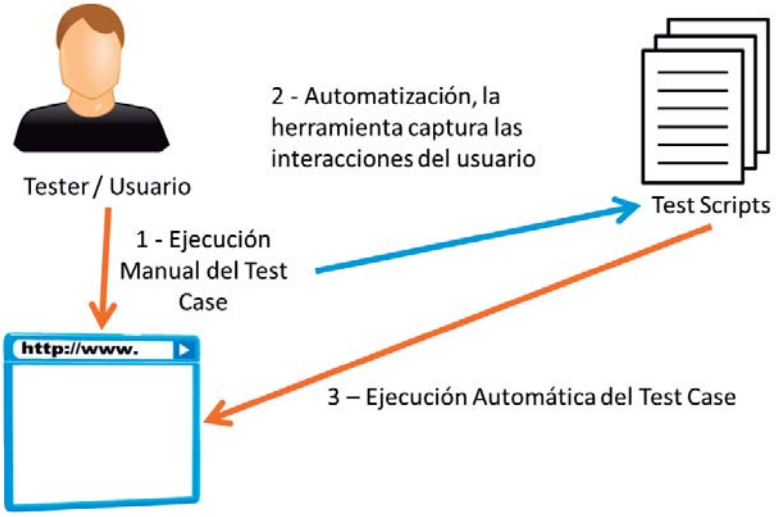


FIGURA 23 - ENFOQUE DE RECORD & PLAYBACK

Los scripts que se generan a partir de la captura de las acciones de usuario generalmente se tienen que modificar, donde 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*), o 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 un 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 la KB si estamos hablando de un sistema desarrollado con GeneXus[28].

El resultado obtenido dependerá de cada herramienta, pero generalmente serán casos de prueba especificados en cierto lenguaje, datos de prueba o scripts de 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. Esto permite trabajar con un mayor grado de abstracción, sin tener que lidiar con los problemas técnicos, concentrándose solo en el modelado del problema, y haciendo 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 relacionados con el ambiente de automatización, considerando que el diseño se hará manualmente, no necesariamente con herramientas que usen técnicas basadas en modelos.

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 me plantee 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.

---

[28] GeneXus: http://www.genexus.com

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[29]:

1. Testing consistente y repetible, asegurando que se ejecutan 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 seguidos (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.

Creemos que si bien hay varias parecidas, 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:

10. Seguir un enfoque de integración continua, y de esa forma tener una detección temprana de defectos. Entonces voy a tener un conjunto de casos de prueba que voy a ejecutar todas las noches.

---

[29] Extraído del libro "*Software test automation: effective use of test execution tools"* de *Mark Fewster & Dorothy Graham*

11. Contar con un conjunto de casos de prueba a ejecutar ante cada nueva versión del producto.
12. Puedo tener pruebas básicas, que me 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.
13. 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 de que un cliente reporta un error, se corrige, y a los dos meses el cliente se queja 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 un tester manual puede 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 están configuradas con https, eso se puede programar y automatizar para que en todas las pruebas se verifique, y en cambio un tester manual tal vez no le presta 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 un tester que sepa programar (generalmente), y es 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 hayan 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 del mismo).
- Acuerdos de nivel de servicio (SLA).
- Dinero o vidas en juego (suena dramático, pero sabemos que hay muchos sistemas que manejan muchas cosas muy sensibles).

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 combinándolo 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.

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 ponerse 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 que a continuación comenzaremos a ver más en detalle este tipo de cosas.

¿Cómo Automatizar?

Consideremos que ya tenemos los casos de prueba diseñados, y 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, comenzaremos revisando el Inventario de Funcionalidades, asignando prioridades a cada una. Luego, asignaríamos 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 que seamos un equipo de varios testers) y para organizarlo, 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” o “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” que digamos que es un identificador válido. Estos últimos son los que realmente podemos ejecutar.

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 agregar más casos de prueba simplemente agregando más datos.

Entonces, antes que nada vamos a grabar el caso de prueba con la herramienta, y como para grabar es preciso ejecutar sobre la aplicación, ahí vamos a necesitar un caso de prueba con datos. El script resultante corresponderá a un caso de prueba con datos, y si lo ejecutamos, ejecutará 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 me 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 cerramos un ciclo de desarrollo y queremos ejecutar todas las pruebas disponibles).

Estos criterios incluso se podrían combinar, teniendo criterios cruzados, o anidados.

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 sé 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*[30], por si quieren investigar más al respecto).

[30] 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 meternos más en el asunto, nos iremos enfrentando a un montón de situaciones más complejas o no tan evidentes de resolver. En esta sección intentaremos mostrar algunos enfoques para atacar muchas situaciones de las 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 queremos 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, redunda 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 considerarlos 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, 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 del mismo. 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, contando qué es lo que 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 Figura 24 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 24). De esa forma podemos reutilizar las pequeñas partes y hacer distintas pruebas que las compongan de distintas formas.

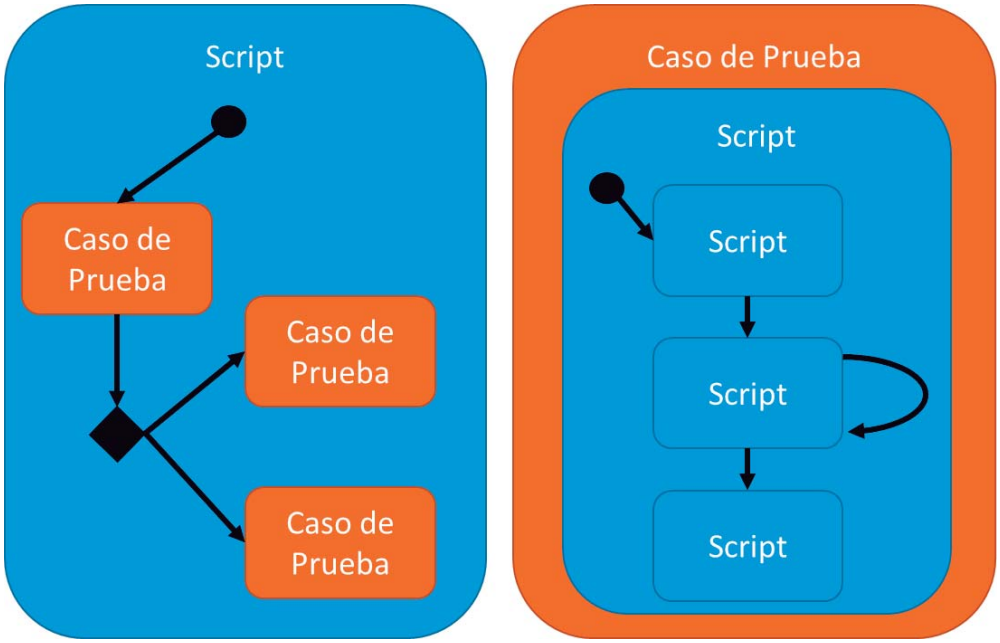


FIGURA 24 - RELACIÓN ENTRE SCRIPTS Y CASOS DE PRUEBA

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

Ventajas

- Mantenimiento: más fácil de mantener, 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 24.
- Análisis de resultados: es más sencillo analizar los reportes de resultados.

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 lineal en caso de que los resultados sean deterministas, y solo si tienen alguna variabilidad no definida de antemano 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 diseño un único caso de prueba que contempla distintas opciones, probablemente sea más difícil de entender, y para eso tenga que analizar qué se está consultando en cada decisión (bifurcación), qué se hace en un flujo u otro, etc., a menos que sea muy cuidadoso y llene 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 agrego 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*. Los que ya han automatizado saben que esto es un problema y a los que 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 Negativo: un estudio indica enfermedad cuando no la hay.
- Falso Positivo: un estudio indica que todo está normal cuando el paciente está enfermo.

Llevándolo a nuestro campo podríamos decir que:

- Falso Negativo: 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 Positivo: la ejecución de una prueba que no muestra un fallo a pesar de que la aplicación tiene un bug. Tanto esto como el falso negativo pueden darse por ejemplo por un estado inicial incorrecto de la base de datos o problemas en la configuración del entorno de pruebas.

O sea, el resultado de que sea negativo es falso, o el resultado de que sea positivo es falso.

Si creemos que el falso negativo nos genera problemas por agregar costos, con un falso positivo ¡hay errores y nosotros tan tranquilos! confiados de 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*.

Por un lado, para bajar el riesgo de que ocurran problemas de ambiente o datos de prueba, deberíamos tener un ambiente controlado, que solo sea accedido por las pruebas automáticas. Ahí ya nos vamos a ahorrar dolores de cabeza y el no poder reproducir problemas detectados por las pruebas, ya que si los datos cambian constantemente, no sabremos qué es lo que pasa.

Por otra parte, ¡deberíamos probar los propios casos de prueba! porque ¿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 los testers para probarlos.

Probando en Busca de Falsos Negativos

Si el software está sano y no queremos que se muestren errores, debemos asegurarnos que la prueba está 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 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 Positivos

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, y 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 tengo de entrada un archivo de texto plano, cambio algo en el contenido del archivo para obligar a que la prueba falle y verifico que el caso de prueba automático encuentra 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 intento 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 me 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, o 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 darle respuesta a estas preguntas veamos lo que se representa en la Figura 25. 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.

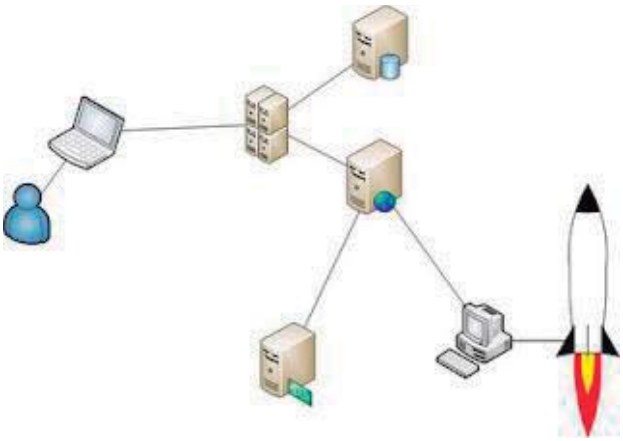


FIGURA 25 - PRUEBAS DE SISTEMAS CON INTERACCIÓN CON SISTEMAS EXTERNOS

Las herramientas de automatización (al menos 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 pude 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, e incluso probar la interacción con el mismo. El artefacto que simula el servicio externo generalmente se lo llama *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 SoapUI[31], que brinda una interfaz muy amigable para generar *Mock Services* así como para testear Servicios Web.

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. Esto está bastante ligado a la herramienta que se elija y las facilidades que brinde.

[31] SoapUI: http://soapui.org

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.

Agreguemos la complicación de que tal vez tenemos distintas pruebas a realizar con distintas configuraciones, parametrizaciones, etc. Entonces para esto tenemos más de un ambiente de pruebas, o tenemos un ambiente y muchos *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 hay cambios en el sistema en los que se modifica 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 fallarán, y no estaremos siendo eficientes con nuestros esfuerzos. Es importante que cada vez que una prueba reporta un fallo, es 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 largo. Se podría tomar la decisión en base a este pseudo-algoritmo:

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

Piensen en 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 hay que considerarlo al planificar. Y además de planificar *cuándo* hay que pensar en *quién*. Generalmente se puede apuntar a tener algunos ambientes bien separados. A modo de ejemplo simple (ver Figura 26):

- el de desarrollo (cada desarrollador)
- el consolidado de desarrollo
- el de testing (dentro del equipo de desarrollo)
- el de pre-producción (testing en instalaciones de prueba del cliente)
- el de producción.

El conjunto de pruebas y la frecuencia de las mismas en cada uno de estos ambientes seguramente sean diferentes.

Por ejemplo, en desarrollo se tiene la necesidad de mayor agilidad, ya que se van a querer correr con mucha frecuencia, deseable que después de cada cambio importante, antes de hacer un *commit* en el repositorio de código. Ahora, para eso sería muy conveniente correr solamente las pruebas necesarias. Estas pruebas tienen el fin de darle un feedback rápido al desarrollador.

Una vez que el desarrollador libera su módulo o cambios a la consolidación, se ejecutarían pruebas a modo de *Pruebas de Integración*. Acá lo ideal sería que corran en las noches, así de mañana cuando llegan los desarrolladores tienen una lista de posibles asuntos a solucionar, y que tengan un *feedback* de los cambios que introdujeron ayer. Mientras menos tiempo pase entre sus cambios y el resultado de las pruebas, más rápido lo van a

solucionar. Esto permitiría evitar que pasen a testing cosas que no funcionen. Serían como pruebas de humo en cierta forma.

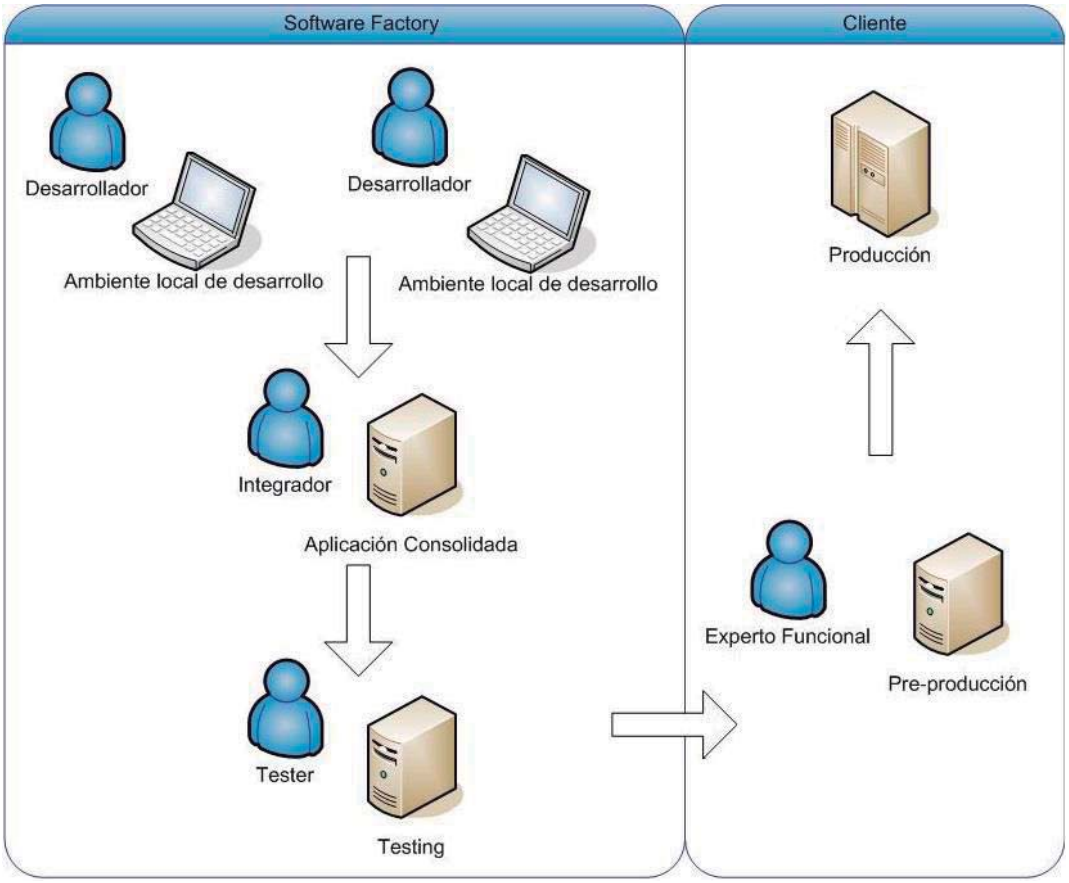


FIGURA 26 - UN POSIBLE ESQUEMA DE AMBIENTES DE PRUEBA

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 han reportado y se han marcado como solucionados no están presentes en esta nueva versión. Este conjunto de pruebas puede tardar más tiempo en ejecutar. Estas pruebas no tienen por qué ser periódicas sino que pueden adecuarse al cronograma del proyecto con las fechas previstas de liberación.

Cuando se llega a la versión entregable de la aplicación (aprobada por el equipo de testing), la misma se libera al cliente. Este por lo general la testea también en un ambiente de pre-producción, 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.

Este conjunto de pruebas podría ser como mínimo el que se corrió 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 Hago con un Bug?

Si un reporte me dice que hubo un fallo, lo primero es determinar si el fallo se debió a los Test Cases. Hay que asegurarse que la culpa no es del test antes de reportar un bug a un desarrollador. Eso mejora las relaciones :) Esto hay que verificarlo manualmente, para ver si se puede identificar la causa, para ver si ocurre siempre, hay que ver si fue un tema de datos, ambiente, casualidad, etc. Luego, hay que ver de 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 el desarrollador, y más fácil para él reproducir esa situación, como para darse cuenta si lo está resolviendo o no.

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

COMENTARIOS FINALES DEL CAPÍTULO

"Test automation is simply an automatic way of doing what testers were doing before" - Steve Rowe (Tester at 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 sea 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 un tester 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! Ese párrafo forma parte de un post[32] en un blog de Steve Rowe, a lo cual James Bach le dio respuesta en su blog[33], 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 hace un tester. 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 sea mucho mejor, en el sentido que al

---

[32] Post de Steve Rowe: http://blogs.msdn.com/steverowe/archive/2007/12/19/what-is-test-automation.aspx

[33] Respuesta de James Bach: http://www.satisfice.com/blog/archives/118

ejecutarlo manual 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 consisten en simular carga en el sistema bajo pruebas para analizar el desempeño del mismo durante la ejecución de la prueba, pudiendo encontrar cuellos de botella y 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 servidor. Para poder simular muchos usuarios con poca infraestructura de 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

Dos aspectos de calidad fundamentales para reducir riesgos en la puesta en producción de un sistema son la funcionalidad y el rendimiento de un sistema.

Para verificar y mejorar la correcta funcionalidad de un sistema se recurre a pruebas a distintos niveles: pruebas unitarias, pruebas de integración, pruebas de sistema. Generalmente los proyectos de desarrollo son iterativos, contando con varias liberaciones del producto a lo largo del tiempo, ya sea a causa de la planificación de liberación de versiones, o por mantenimientos realizados luego de la puesta en producción. Es por esto que resulta necesario realizar pruebas de regresión (verificando en cada liberación que el software no tiene regresiones), y para esto generalmente se suele apoyar en herramientas de automatización de pruebas, para ejecutar estas pruebas de regresión. Este tipo de cuestiones son las que hemos venido apuntando en los capítulos previos.

Por otra parte, para las pruebas de rendimiento se utilizan también herramientas para simular la carga que generan múltiples usuarios conectados concurrentemente. Mientras se ejecuta la carga se analiza el desempeño de la aplicación en busca de cuellos de botella y oportunidades de mejora. De esta forma 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?
- Etc., etc., etc…

Una prueba de rendimiento se define como una investigación técnica para determinar o validar la velocidad, escalabilidad y/o características de estabilidad de un sistema bajo prueba para analizar su desempeño en situaciones de carga[34].

Las pruebas de rendimiento son sumamente necesarias para reducir riesgos en la puesta en producción, logrando así analizar y mejorar el rendimiento de la aplicación y los servidores, al estar expuestos a usuarios concurrentes. Para ello se utilizan herramientas específicas (llamadas *generadoras de carga*) para simular la acción de múltiples usuarios concurrentes. Dentro de las generadoras de carga *open source* más populares se encuentran OpenSTA[35] y JMeter[36].

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 de prueba, se pasa a ejecutar y analizar los resultados. Si bien vamos a ver muchos puntos importantes de estas pruebas, recomendamos principalmente dos lecturas introductorias:

- Un artículo publicado por el Centro de Ensayos de Software, escrito por Federico Toledo, Gustavo Vázquez, Matías Reina, Simón de Uvarow, Edgardo Greisin, Horacio López, llamado *Metodología de Pruebas de Performance*. Este artículo lo presentamos en las *Jornadas Chilenas de Computación* dentro del *Encuentro Chileno de Computación*, en Punta Arenas, Chile, en el año 2008.
- Una serie de artículos publicados por Scott Barber llamados *User Experience, not Metrics*, del año 2001.

Hay distintos tipos de pruebas de performance: load test (test de carga), donde la idea es simular la realidad a la que estará expuesto el sistema cuando esté en producción; stress test, donde se va más allá de la carga esperada como para ver dónde se "rompe" el sistema; endurance (resistencia), para ver cómo se desempeña el sistema luego de una carga duradera por un período largo de tiempo, etc.

---

[34] Meier, J., Farre, C., Bansode, P., Barber, S., Rea, D.: Performance testing guidance for web applications: patterns & practices. Microsoft Press (2007).

[35] OpenSTA: http://opensta.org

[36] JMeter: http://jmeter.apache.org

Cualquiera que sea el tipo de prueba de performance que se esté haciendo, hay toda una preparación necesaria que lleva 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[37], 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 son OpenSTA y JMeter). A esta tarea se la suele denominar como automatización o robotización. Esto, por más que estemos utilizando herramientas de última generación, costará un tiempo que separa más el día de inicio del proyecto del día en que se comienzan a reportar datos sobre el desempeño del sistema a nuestro cliente.
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 la gente no tiene en cuenta temas relacionados con el rendimiento del sistema cuando esté en producción. También pasa que se subestima la importancia y el costo de las pruebas. ¿Quién no ha oído estas opciones al hablar de pruebas de performance?

- *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

[37] *Ramp-up*: Velocidad y cadencia con la que ingresan los usuarios virtuales al inicio de una prueba.

haya casos en los que la performance no requiera tanto esfuerzo para evaluar (un sistema que sea monousuario, que sea accedido por 2 o 3 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 preparamos las pruebas de performance, impidiéndonos automatizar, u obteniendo errores al ejecutar pruebas.
2. Que ajustemos el sistema para que su performance sea buena, y luego en las pruebas funcionales se detectan problemas, que implican 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 ir ejecutando pruebas intermedias paralelas con el desarrollo. Se podrían ejecutar pruebas anticipadas sobre un módulo o funcionalidad específica, apenas se cuente con una versión estable de la misma; el propio desarrollador podría ejecutar sus propias pruebas de performance en menor escala, etc. De esta forma se llega a las pruebas finales con una mejor calidad, bajando la probabilidad de tener que realizar grandes cambios.

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, el escenario de carga (los casos de prueba y la mezcla de usuarios ejecutándolos), 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:

- **Pasaremos a una nueva infraestructura**. Quizá nuestro sistema ya está funcionando en determinada plataforma, y como queremos cambiar la plataforma, o pasar a otro equipo más potente (o más económico), o necesitamos cambiar un componente o reestructurar la red interna de la empresa.
- **Estamos por liberar una nueva versión del sistema**. Puede tratarse de nuevas funcionalidades o cambios en funcionalidades existentes.
- **Cambio de arquitectura**. Quizá esta es una de las situaciones más comunes con las que nos hemos enfrentado, 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, o 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 accediendo al sistema aumentará considerablemente. Esto puede ser también pensando en una fecha especial, tal como navidad, 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 garantizado la validez de nuestro nuevo escenario.

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 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. Generalmente, 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**. Está permitido preguntar a los responsables de infraestructura, e incluso a los desarrolladores, pues ellos tendrán idea de cómo programaron cada cosa.
- **Cuáles son las más riesgosas para la operativa del negocio**. Está permitido preguntarle al que pierde dinero si algo no funciona. También a los usuarios.
- **Cuáles son los más usados**. Está permitido 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.

Tenemos 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*.

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, que generalmente se les llama "*think time*".

No todos los usuarios ejecutan las funcionalidades de la misma forma. Generalmente podrán seguir distintos flujos, elegir distintos parámetros, distintas opciones, ejecutando 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, si analizando el uso del sistema vemos que hay 200 usuarios que realizan extracción por cajero y unos 20 que realizan pagos con tarjeta en comercio, para el sistema es la misma interfaz y la operación es muy similar, entonces podríamos considerar que tenemos 220 usuarios ejecutando extracciones. Esto nos reduce el costo a la mitad y nos da un beneficio similar, o sea, la prueba no será exacta, perfecta, pero será suficientemente buena. No decimos que esto haya que hacerlo siempre, sino cuando no hay tiempo 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 carga 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 esa mezcla que caracteriza la carga esperada del sistema en un momento determinado (en la operativa diaria en 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 ¡los 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 y en el mes de mayor carga, etc. Pueden querer simularse 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**.

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 recibe por parte de los usuarios reales como por parte de los usuarios virtuales. En la Figura 27 se muestra un ejemplo de un escenario real y un escenario simulado que deberían ser equivalentes para el punto de vista del sistema.

FIGURA 27 - ESCENARIOS EQUIVALENTES DE USUARIOS REALES Y USUARIOS VIRTUALES

Para ver un ejemplo de lo que es un escenario de carga definido directamente en una herramienta de simulación de carga (que en definitiva es lo que queremos), veamos la Figura 28. Ahí se muestra la consola de OpenSTA y una prueba de ejemplo definida con tres casos de prueba (TC1_AddProduct, TC2_AddClient y TC3_AddInvoice). Estos se ejecutan concurrentemente, uno con 200 usuarios concurrentes, otro con 100 y otro con 15. La ejecución de todos estos casos de prueba en conjunto, en forma concurrente, está simulando la realidad de ese supuesto sistema.

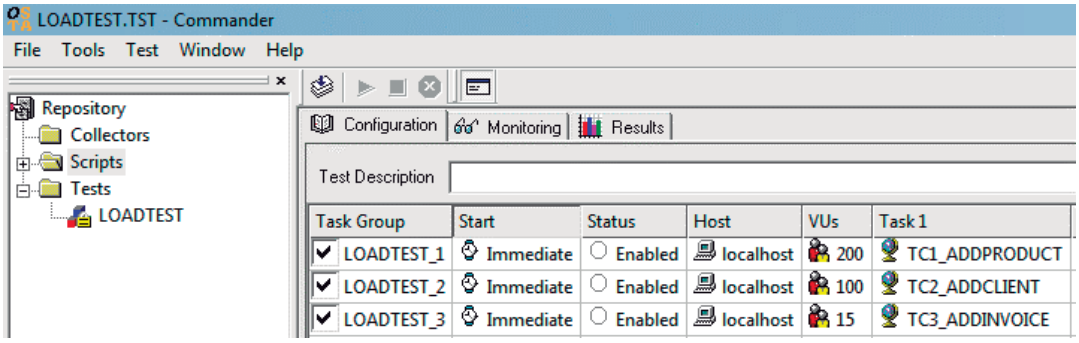


FIGURA 28 - ESCENARIO DE CARGA MODELADO EN OPENSTA

La pregunta aquí es ¿por qué no consideramos procesos estocásticos, y modelar los usuarios de un sistema como un proceso de distribución de Poisson[38]? Quizá sea lo más lógico desde

[38] Proceso de Poisson: http://es.wikipedia.org/wiki/Proceso_de_Poisson

un punto de vista académico. Creemos que es muy interesante hacer un análisis y ver por qué típicamente en las herramientas de simulación de carga no se consideran los escenarios de carga con mayor complejidad:

- **Queremos pruebas repetibles**. Nunca una ejecución puede ser exactamente igual a otra, pero queremos intentar reproducir las mismas situaciones. 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 para 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). Pero estas variables solamente pueden obtenerse realizando simulaciones de carga reales. Lograr llegar a 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, para este tipo de pruebas, es mejor en cuanto a costo/beneficio.

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. Si utilizamos la base de datos de pruebas, seguramente no detectemos las consultas SQL que sean poco eficientes ya que al trabajar con pocos registros la ineficiencia pasa inadvertida.

Entonces, intentaremos usar los servidores de producción (si no están en uso), e intentaremos usar la base de datos de producción. Si hay problemas de confidencialidad de datos, en estos casos tendremos que analizar la factibilidad de enmascarar esos datos, o de generar una base de datos de prueba artificial pero con volumen 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, así sabremos qué componentes monitorizar, y luego ver qué indicadores recogeremos, con qué herramientas, y cuáles son los umbrales aceptables.

Para 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, JVM, .NetFramework, 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:

- Los integrantes del equipo se van familiarizando 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.

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; por otro lado tendremos los **indicadores de segundo nivel**, cuyo objetivo es obtener más información y 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.

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. 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 al promedio de los tiempos de respuesta, o al 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 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, o 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, al entrar 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 necesitan procesar 2500 facturas en una hora.
- En las cajas se atienden 2 clientes por minuto en cada caja.

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 puede entrar y comprobar la velocidad de respuesta del sistema, viendo si está usable o insoportablemente lento. Por otro lado, estos resultados hay que cruzarlos también con los consumos de recursos, pero eso en cualquier caso también. 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.

Un concepto explicado por Harry Robinson[39] (el precursor del *model-based testing)* 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*[40], indicando que se podría probar calcular el camino de ida de *A* hacia *B*, y luego de *B* hacia *A*, y 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.

[39] Harry Robinson: http://www.harryrobinson.net/

[40] Google Maps: http://maps.google.com

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 de *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 el mismo. 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, obteniendo 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 procesos 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.

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 (HTTP, SIP, SOAP, ISO8583, 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, un script que llevaría 4 líneas en Selenium se corresponde con un script de 848 líneas en OpenSTA. 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 como puede verse en el ejemplo de la Figura 29. Embebidos viajan parámetros, cookies, variables de sesión, y todo tipo de elementos que se utilicen en la comunicación con el servidor. El ejemplo de la figura corresponde al *HTTP Request* primario del último paso del caso de prueba, por lo que se puede ver que se envía el valor "computer" en el parámetro "vSEARCHQUERY" (en el recuadro rojo).

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, debiendo ser únicas por usuario. Se deberán ajustar parámetros, tanto del encabezado como del cuerpo del mensaje, etc.

```
!Cookie Header:
!>  __utma=188893609.1589001788.1311037720.1311037720; ASP.NET_SessionId=fhhjz145d4l1me55cp0uwtit
POST URI "http://localhost/sampleapplication/search.aspx HTTP/1.1" ON 1 &
    HEADER DEFAULT_HEADERS   &
    ,WITH {"Accept: */*", &
        "Accept-Language: en-US", &
        "Referer: http://localhost/sampleapplication/search.aspx", &
        "gxajaxrequest: 1", &
        "Content-Type: application/x-www-form-urlencoded", &
        "Content-Length: 2254", &
        "Connection: Keep-Alive", &
        "Pragma: no-cache", &
        "Cookie: "+S_cookie_1_0+"; "+S_cookie_1_1}       &
,BODY {"vSEARCHQUERY=computer&BUTTON1=Search&GXState=%7B%22_EventName%22%3A%22E~<27>SEARCH~<27>" &
    ".%22%2C%22_EventGridId%22%3A45%2C%22_EventRowId%22%3Aundefined%2C%22MPW0027%22%3A%22RecentLinks" &
    "%22%2C%22MPW0027AV6FormCaption_PARM%22%3A%22Search%22%2C%22MPW0027AV7FormPgmName_PARM%22%3A%22S" &
    "earch%22%2C%22MPW0027_CMPPGM%22%3A%22recentlinks.aspx%22%2C%22MPW0027nRC_Links%22%3A%222%22%2C%" &
    "22MPW0027wcpOAV6FormCaption%22%3A%22Search%22%2C%22MPW0027wcpOAV7FormPgmName%22%3A%22Search%22%" &
```

Figura 29 - Script de Prueba de Rendimiento con OpenSTA

En base a nuestra experiencia en más de 8 años realizando pruebas de rendimiento, la construcción de estos scripts ocupa entre el 30% y el 50% del total de esfuerzo invertido en las pruebas de rendimiento. Por otra parte, el mantenimiento de estos scripts suele ser tan complejo que 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.

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. Con OpenSTA hemos podido generar una carga de 1500 usuarios virtuales utilizando tan solo 5 máquinas, las cuales reportaron estar utilizando una cantidad acotada de recursos.

A pesar de que OpenSTA se encuentra discontinuada, la seguimos prefiriendo dentro de las herramientas open source. Existen otras alternativas muy buenas también, como por ejemplo JMeter, la cual en cambio es un proyecto sumamente activo, y que es útil además para muchos más protocolos y no solo para HTTP como OpenSTA. Sin embargo, en nuestra experiencia al menos, siempre hemos visto un menor rendimiento de JMeter. O sea, con una misma máquina para simular pruebas hemos sido capaces de simular menos usuarios en una máquina con JMeter que en una máquina con OpenSTA.

Preparación de la Infraestructura de Pruebas

De este punto solo 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 mismos scripts de pruebas automáticas. 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 considerando que una ejecución modifica el estado de los datos de la base de datos, deberemos recuperar el estado inicial de la misma. Es por este motivo que es 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, preparar scripts o programas para procesar los datos. Muchas veces será muy útil preparar alguna planilla de cálculo en la cual se carguen los datos y ya nos prepare las gráficas, en otros casos deberemos *parsear* los resultados y cargarlos en algún sistema de análisis. En cualquier caso, es conveniente automatizar la recolección de métricas con las herramientas definidas y así facilitar su análisis.

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 éste 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 ésta 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 Figura 30 consta de una representación que utilizaremos para explicar esta situación.

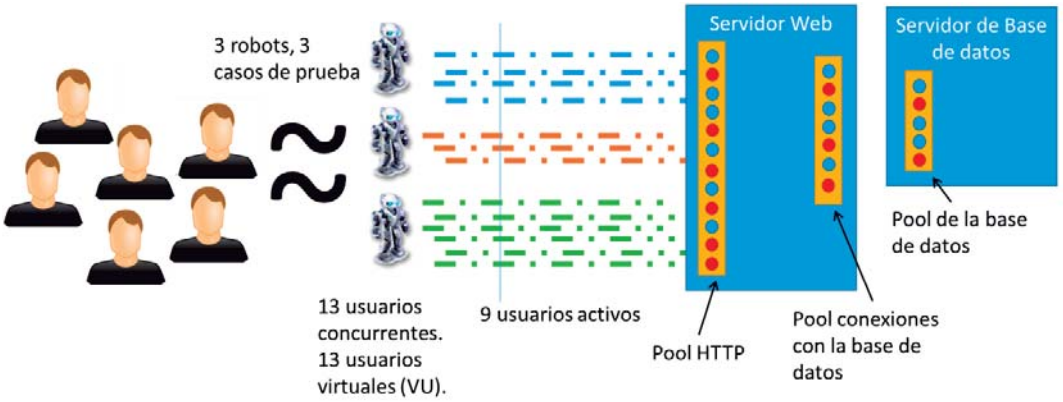


FIGURA 30 - USUARIOS CONCURRENTES, VIRTUALES, ACTIVOS

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, habrán tantos "usuarios virtuales" como "usuarios concurrentes" o "usuarios reales" habrán 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, 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 avanzamos se comienza a probar variantes, ejecutar una prueba, y se cambia otra cosa, y se vuelve a ejecutar otra prueba, etc. Además, muchas de las pruebas que ejecutamos no son válidas, porque nos olvidamos de activar algún log, de reiniciar cierto componente, de 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 tenerla 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 se basa 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 para ese caso de prueba, 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 como comparación de rendimiento ante carga y rendimiento *stand-alone*.

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 están funcionando bien antes de lanzar la carga objetivo. Si ejecutamos el 100% de la carga al comienzo seguramente 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 Figura 31 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 que alguna oportunidad de mejora se

detecte debería implementarse y para verificar que realmente se logra 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, y por esto es necesario verificarlo empíricamente antes de continuar. Una vez que estos cambios se implementaron y se verificó su mejora, entonces ahí 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 hasta qué punto soporta la infraestructura actual.

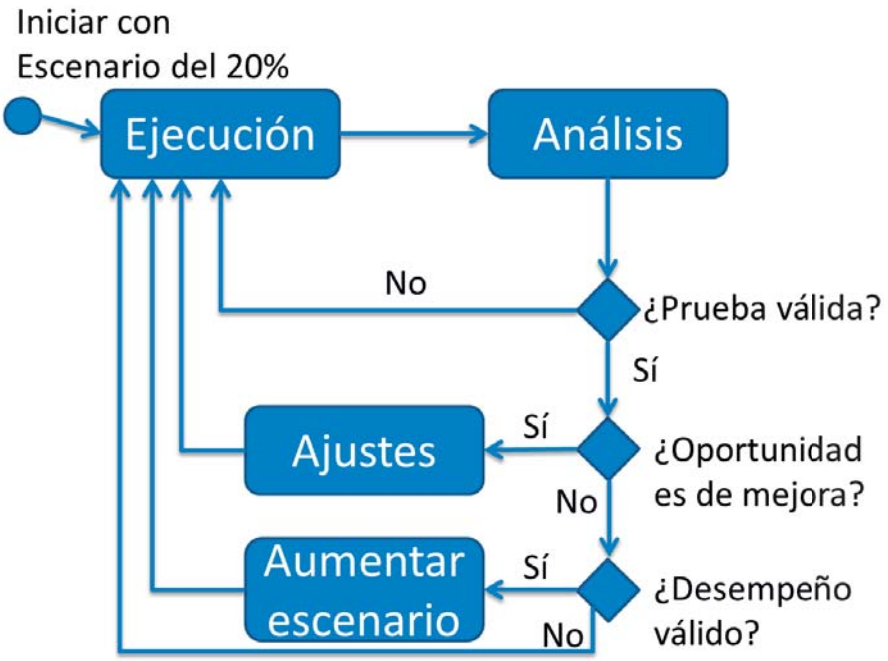


FIGURA 31 - WORKFLOW PARA GUIAR LA EJECUCIÓN

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 Figura 32) 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 inicia, y a partir de ese momento el cliente comienza a recibir resultados. Es decir: desde que el cliente firmó el contrato 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 duran 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, además, estaremos verificando la validez de nuestras pruebas, veremos si nos estamos enfocando a lo que nuestro cliente necesita, dando el tipo de resultados que espera obtener. Si tenemos algún error en el análisis o diseño de las pruebas, y nos enteramos de esto al dar los resultados, eso 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.

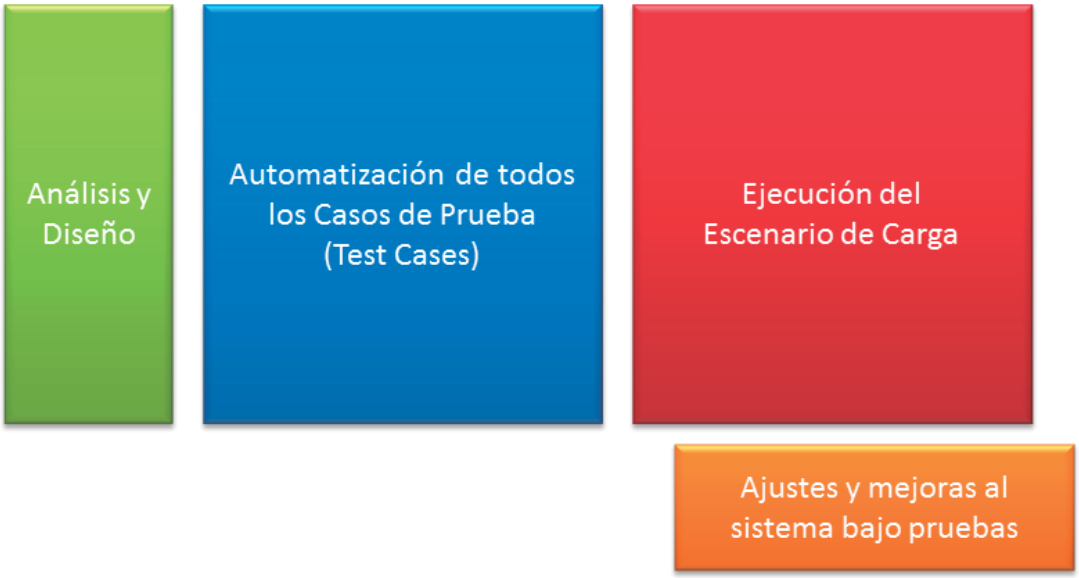


FIGURA 32 - ENFOQUE DE EJECUCIÓN TRADICIONAL DE PRUEBAS DE PERFORMANCE

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.

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 el performance del sistema.

Pipeline entre Automatización y Ejecución

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 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 (ver Figura 33). 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.

Entonces, el pipeline que nos daría más agilidad para obtener resultados lo antes posible en la automatización de pruebas, aprovechando ejecutar pruebas iniciales con cada nuevo caso de prueba que se automatice.

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.

FIGURA 33 - PROCESO AGILIZADO DE PRUEBAS DE PERFORMANCE

¿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 es 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 externo para que hagan las pruebas de sistema.

Nosotros 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. Mientras eso sucede, podríamos mirar la base de datos, qué tiempo tuvieron las SQLs, si utilizaron los índices correctamente, qué pasa si crece el tamaño de esa tabla, etc. Luego, ese programa en lugar de simplemente ejecutar muchas veces un método, podría levantar múltiples procesos que ejecuten muchas veces ese método (siempre teniendo en mente lo que sucederá cuando esté en producción). Podríamos ver qué tanta memoria consume, cómo se maneja el pool de conexiones, si observamos bloqueo entre tablas, etc.

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 tienen en cuenta estos aspectos en forma temprana, desde el desarrollo.

Algunas ventajas de este enfoque:

- Los desarrolladores se comprometen con la performance, verifican y no liberan módulos con problemas de performance graves.
- Los desarrolladores se familiarizan con el uso de recursos de sus sistemas, descubren qué cosas de su código generan problemas y seguramente les ayude a evitarlo a futuro.
- Menos riesgo de tener problemas de performance.
- Los problemas se detectan antes, con lo cual será más barato resolverlos.

Y a su vez podríamos listar algunas desventajas:

- Más tiempo para liberar la primera versión.
- Necesidad de aprender a utilizar herramientas de monitorización.

Claro que esto no es para hacerlo con cada método 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 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 voy 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*, o 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 cuenten 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 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, y 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 aplicaciones que están en producción en el mismo servidor.

De todos modos esta aproximación tiene una gran ventaja: nos aseguramos que la infraestructura de prueba es similar a la de producción, en realidad, 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, y 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, e 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.

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 la técnica de *benchmark*. Antes de hacer pruebas de performance es necesario conocer el rendimiento y límite de sus componentes a efectos de tener información base sobre la cual comparar en el momento de realizar las pruebas de performance. Ejecutar los *benchmarks* frente a los componentes permite además detectar posibles desvíos o problemas en los mismos, lo cual ayuda a eliminar una posible causa de "ruido" en las pruebas de performance, es por ese motivo que se recomienda ejecutar de forma previa a los test de performance para solventarlos lo antes posible 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 o 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 te 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 que 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 o 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 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 (tunning) 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 o 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 pueden 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 seguirán siendo igual de frágiles. 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).

Otro enfoque válido 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.

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 Java que se pueden encontrar aquí: http://java-source.net/open-source/profilers.

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 porque el 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, y 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 *caché 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, enfocándose en probar unitariamente y no concurrentemente diferentes actividades. Quizá probamos una operación de “extracción de dinero” con 1000 usuarios, y por otro lado una de “depósito de dinero” con 1000 usuarios, y como en total no llegaremos a tener más de 500 entonces nos quedamos tranquilos, 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 está 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, y 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, 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.

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"[41], 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, y 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?
- 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.

[41] Doctor House (2004-2012): famosa serie televisiva http://www.imdb.com/title/tt0412142/

- ¿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 eviten los síntomas presentados. De esta forma, con esta prueba, solucionamos el problema que detectamos. De otro modo, los usuarios hubiesen sufrido esas demoras, y el experto de base de datos hubiese tenido muchas quejas, y tendría que haber ajustado estos parámetros en producción con los riesgos e impactos que esto pueda tener.

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 DE UN TESTER

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.

¿Qué Habilidades Necesita Desarrollar un Tester?

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 el tester. 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 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 conmigo mismo a ver qué voy a decir y ahora él 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 interrumpir, aún si nos estuvieran contando un problema y a la mitad del mensaje se nos ocurre una solución, puede 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: manteniendo 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 ser claro al hablar, ser específico 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 un tester! 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. Para mejorar más rápidamente 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 RAE[42] 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* 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 hayan ambigüedades, si está completo, etc.

---

[42] RAE: Real Academia Española, http://www.rae.es/

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 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.

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 utiliza el teléfono o 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 donde el equipo de desarrollo conoce profundamente el producto, y entonces puede parecernos que no 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.

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, y creemos que muchas veces es porque los testers que venimos de formación informática tenemos "miedo" que nos "roben" el trabajo los expertos de otras áreas, y he escuchado 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

Tener la capacidad de trabajar 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. 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.

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 un tester *debe* tener conocimientos técnicos –o no– es un tema siempre abierto a debate. Según una polémica presentación[43] de James Whittaker, el testing funcional tradicional estaría en vías de extinción y por lo tanto ha planteado como imprescindible orientarnos hacia un perfil técnico o correremos 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 un tester 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?

---

[43] James Whittaker's STARwest Keynote (http://www.testthisblog.com/2011/10/james-whittakers-starwest-keynote.html)

¿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 esta tarea en alguien que conozca de la herramienta y 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 los testers, que si usamos pruebas automatizadas no significa que los vamos 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, haciendo que nuestro perfil se mueva en las distintas zonas del conocimiento planteadas en la Figura 34. 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.

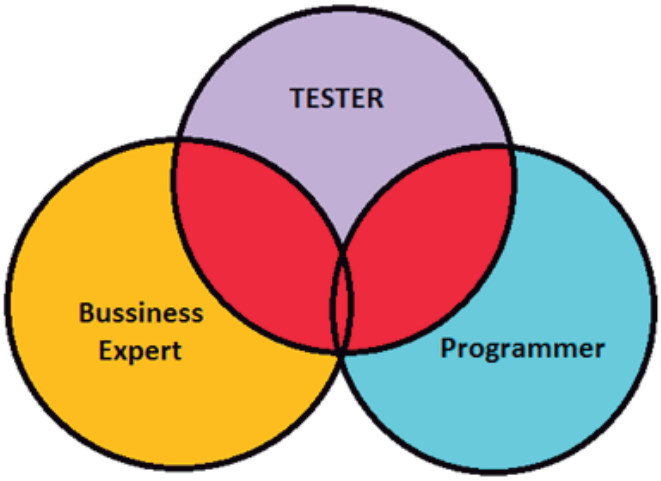


FIGURA 34 - CONOCIMIENTOS DE UN TESTER

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 leyendo material de metodología y experiencias de pruebas automatizadas en general. Para comenzar, una lectura recomendada, entre tantas otras, es la 4ta Edición de la revista *Testing Experience*[44] dedicada al testing automatizado.

[44] Revista *Testing Experience*: http://www.testingexperience.com/

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, angustiados 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 hacen 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.

La ciencia se compone de errores, que a su vez, son los pasos hacia la verdad.

– Julio Verne

El ingeniero Federico Toledo, PHD en Informática en la Universidad de Castilla-La Mancha, es un especialista en testing y cuenta con una destacada trayectoria privada, institucional y académica en Uruguay (su país de nacimiento) y Europa.

Es socio fundador e integrante permanente del directorio de ABSTRACTA, empresa que provee productos y servicios de testing a compañías en Estados Unidos y América Latina, y es un blogger frecuente de artículos especializados en testing en lengua castellana.

Reseña

Introducción a las Pruebas de Sistemas de Información es una de las primeras publicaciones en español sobre aspectos prácticos de testing. En el mismo se abordan todas las dimensiones del testing, mostrando su propósito y beneficio. Integrando técnicas modernas de forma dinámica y práctica le permite al lector realizar una recorrida nutrida y clara sobre el diseño de pruebas, pruebas automatizadas, pruebas de performance y aspectos humanos que debe desarrollar un buen tester.

El público objetivo de *Introducción a las Pruebas de Sistemas de Información* son las personas que trabajan cotidianamente haciendo testing y se enfrentan a un sinnúmero de desafíos y dificultades. Ellos contarán a partir de hoy con una guía de apoyo para enfrentar distintas actividades que realizan, especialmente las relativas a técnicas de diseño de testing, testing automatizado y de performance.

Introducción a las Pruebas de Sistemas de Información está escrito en un lenguaje llano y ameno, y fue pensado y redactado para que su lectura permita la rápida incorporación de conceptos, técnicas y nuevas habilidades para sus lectores. El aprender y disfrutar aprendiendo fue uno de los intereses mayores que sostuvo Federico a la hora de estructurar y dar contenido a su obra.

Auguramos que *Introducción a las Pruebas de Sistemas de Información* sea una excelente experiencia de lectura y aprendizaje que efectivamente eleven al lector a un mayor nivel de conocimiento conceptual y práctico. ¡Ahora, a disfrutarlo!

Lic. Gonzalo Acuña

abstracta
www.abstracta.com.uy

9 789974 993853