Introducción a las Pruebas de Sistemas de Información

***

El primer libro de testing escrito en español

Autor: Federico Toledo

Abstracta, Montevideo, Uruguay, 2020

Prólogo

Junto a la Dra. Beatriz Pérez Lamancha, tuve la suerte de codirigir, en la Universidad de Castilla-La Mancha, la tesis doctoral de Federico Toledo.

Creo que durante los años en que elaboró "Madinga: a methodology for automation testing integrating functional and non-functional aspects" aprendimos muchos los tres: yo, desde luego, al sumergirme junto a él en aspectos de pruebas no funcionales y en la utilización de casos de prueba funcionales para derivar casos de prueba de rendimiento, estrés, etcétera.

Por tanto, el texto que el lector tiene ante sus ojos procede de un auténtico experto en pruebas no sólo a nivel técnico sino a nivel teórico. En este texto, Federico, explica de manera clarísima los conceptos más importantes sobre testing, profundizando después en sus aspectos más prácticos.

Como profesor de Ingeniería de Software en la universidad, creo que este es y debe ser un texto de referencia para los estudiantes de esta materia.

Macario Polo Usaola

Ciudad Real, España.

A QUIÉN ESTÁ DIRIGIDO ESTE LIBRO

Este libro busca ayudar a aprender las bases del testing de forma amena. Está dirigido tanto para aquel que no tiene ningún conocimiento sobre el tema, como para aquel que ya conoce e incluso ya está trabajando como tester ya que cuenta con contenidos como para profesionalizar y profundizar sobre diversas temáticas.

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

En las páginas que siguen compartiremos parte del conocimiento que hemos ido adquiriendo en estos años trabajando en la industria del testing, complementado con investigación sobre temas específicos que me gustaría que puedan aprender. El objetivo final es que el libro genere un impacto en la vida del lector, ya sea accediendo a oportunidades laborales en el ámbito tecnológico, como dando la posibilidad de desempeñarse de mejor manera, pudiendo contar con un libro de referencia para los temas que se necesitan aplicar en el día a día.

Luego de leer este libro se podrá entender qué es la calidad del software y cómo un tester puede colaborar en ella, así como los principales desafíos que hay al respecto junto a posibles soluciones. También se podrá tener un manejo sobre distintas técnicas de testing, ya que el libro cuenta con ejercicios y ejemplos prácticos para aplicar. Es así que se podrá aprender sobre herramientas avanzadas que permiten mejorar las actividades de testing gracias a la automatización aplicada a distintos fines, como las pruebas de regresión automáticas y las pruebas de performance simulando el acceso de cientos de usuarios de manera simultánea.

Más allá de lo noble que pueda ser el fin, lo importante también es el camino, así que lo que espero es que todos puedan disfrutar de la lectura mientras aprenden más sobre este apasionante tema del testing de software.

Sobre la Tercera Edición

Este es un libro con un carácter divulgativo. Queremos plasmar conceptos, buenas prácticas, técnicas y metodologías que hemos investigado, que nos han resultado útiles y creemos que le pueden resultar útiles a otros también. Hablo en plural, porque más que un único autor, siento que represento a un equipo de trabajo. Nada de lo que se presenta aquí es completamente propio, y al mismo tiempo todo lo presentado aquí es propio, pues lo hicimos parte de nuestra cultura de testing, adaptándolo según nuestros criterios y necesidades.

Como se trata de un libro introductorio, tampoco estamos siendo ambiciosos de querer tener algo completamente extenso, sino que priorizamos los conceptos, técnicas y métodos que nos resultan de mayor prioridad (hasta en la redacción de este libro estamos aplicando conceptos de testing: no exhaustivo, priorizando lo importante).

Los que nos conocen, los que han visto charlas nuestras o quienes leen algunos de los blogs en los que participamos (hasta hace unos años el blog de Abstracta en español[1] que hoy está discontinuado, donde escribo mayoritariamente hoy que es mi sitio web[2] en español, o en inglés[3]) 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).

La primera edición fue en el 2014, donde imprimimos 500 copias que se terminaron de vender y regalar en el 2018. En el 2015 imprimimos una segunda edición en México (con cambios menores) como parte de un proyecto que hoy es Abstracta Academy[4], el cual originalmente lo habíamos enfocado en ese mercado. Esta **tercera edición** cuenta con una revisión y varios ajustes, que surgen de algunas nuevas visiones que hemos adoptado en estos 6 años, así como cuidados en el lenguaje que tratamos de mejorar día a día. Hizo falta hacer algunos cambios en ejemplos basados en herramientas que quedaron un poco viejas. También, hemos ajustado algunas cosas en base al feedback recibido en las primeras ediciones, lo cual ha sido lo más valioso de este proceso. Por ejemplo, todos los links que aparecen en el libro, los podés ver en este post, para mayor facilidad:

---

[1] Blog de Abstracta en español (discontinuado): blog.abstracta.com.uy
[2] Sitio web personal: www.federico-toledo.com
[3] Blog de Abstracta en inglés: www.abstracta.us/blog
[4] Abstracta Academy: https://abstracta.academy

Agradecimientos

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!

En particular quisiera agradecer a Giulliana Scuoteguazza y a Andrés Curcio, quienes me ayudaron a escribir un capítulo cada uno en la primera edición del libro. Sin su contribución el libro no contaría con la variedad de temas y puntos de vista. Para esta tercera edición en particular, varios colegas y amigos de Abstracta han aportado en diferentes revisiones y sugerencias, agradezco entonces principalmente a Andrei Guchín, Verónica Gamarra, Alejandro Berardinelli y Sebastián Lorenzo.

También una mención especial para **Macario Polo Usaola**, quien en su momento fue el tutor de mi tesis, y en esta oportunidad colaboró escribiendo el prólogo.

Por último, pero no por ello menos importante, un reconocimiento **a mi esposa Ale**, quien me "hace el aguante" a diario en esta cruzada de evangelizar con el testing y la calidad, ya sea con Abstracta, en la Universidad donde estuve trabajando, con los meetups y eventos que organizo, con las charlas y cursos que doy y artículos o libros que escribo. Ella siempre presente, apoyando e incluso colaborando desde su visión científica. La mejor compañera que me he encontrado en la vida.

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[5]. 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 problema es cuando estos errores producen (y hay miles de ejemplos de que han sucedido) 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[6]. 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 tomaba 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.

Según cuenta el artículo de El País, el sistema tiene 86.000 líneas de código ADA[7] (definitivamente no es ningún “*hello world*”[8]). 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,

---

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

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

[7] Lenguaje de programación ADA:
https://es.wikipedia.org/wiki/Ada_(lenguaje_de_programaci%C3%B3n)

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

pero no se han manifestado o no son lo suficientemente importantes para que los usuarios se vean afectados. Eso es lo importante. ¿Y cómo ayudamos a que se genere software de calidad suficiente? Obviamente, lo que nosotros vamos a decir es que 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 o que son más importantes para el negocio. Esto estará limitado o potenciado por otros factores más relacionados al mercado, pero nosotros nos centraremos en cómo "asegurar la calidad"[9] optimizando los costos lo mejor posible, dejando esas otras discusiones un poco al margen.

Esta visión es un poco más feliz que la que siempre escuchamos de Dijkstra[10], 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.

---

[9] "Asegurar la calidad" va entre comillas a propósito. ¿Alguien puede asegurar la calidad? Hay un artículo de Michael Bolton muy interesante al respecto, indicando que el que quizá pueda hacer eso es el CEO de la compañía, el que maneja todos los recursos y asignaciones, el que toma las grandes decisiones. Nosotros como testers, lo máximo que podemos hacer es dar la información oportuna para que esas decisiones se tomen en mejores condiciones. Estamos muy alineados con su visión:
http://www.developsense.com/blog/2010/05/testers-get-out-of-the-quality-assurance-business/

[10] Edsger Dijkstra: https://es.wikipedia.org/wiki/Edsger_Dijkstra

La Calidad, El Testing y sus Objetivos

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

Calidad

Una pregunta que resulta muy interesante es la que cuestiona ¿qué es la calidad? Existe una ley divina que indica que todo curso, seminario, tutorial o materia electiva de Facultad de Ingeniería relacionado con testing debe iniciar haciendo esta pregunta. 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, 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[11]. 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".

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

---

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

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 al proceso de generar un producto de calidad. ¿Cómo? Detectando posibles incidencias de diversa índole que pueda tener cuando esté en uso, detectando riesgos, haciendo preguntas (incómodas muchas veces), empatizando con ese posible usuario.

Objetivos

La forma en la que estaremos "aportando calidad" será principalmente buscando fallos[12]. 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 también estamos aportando información relacionada a la calidad. Entonces, el equipo tendrá más confianza en que los usuarios encontrarán menos problemas.

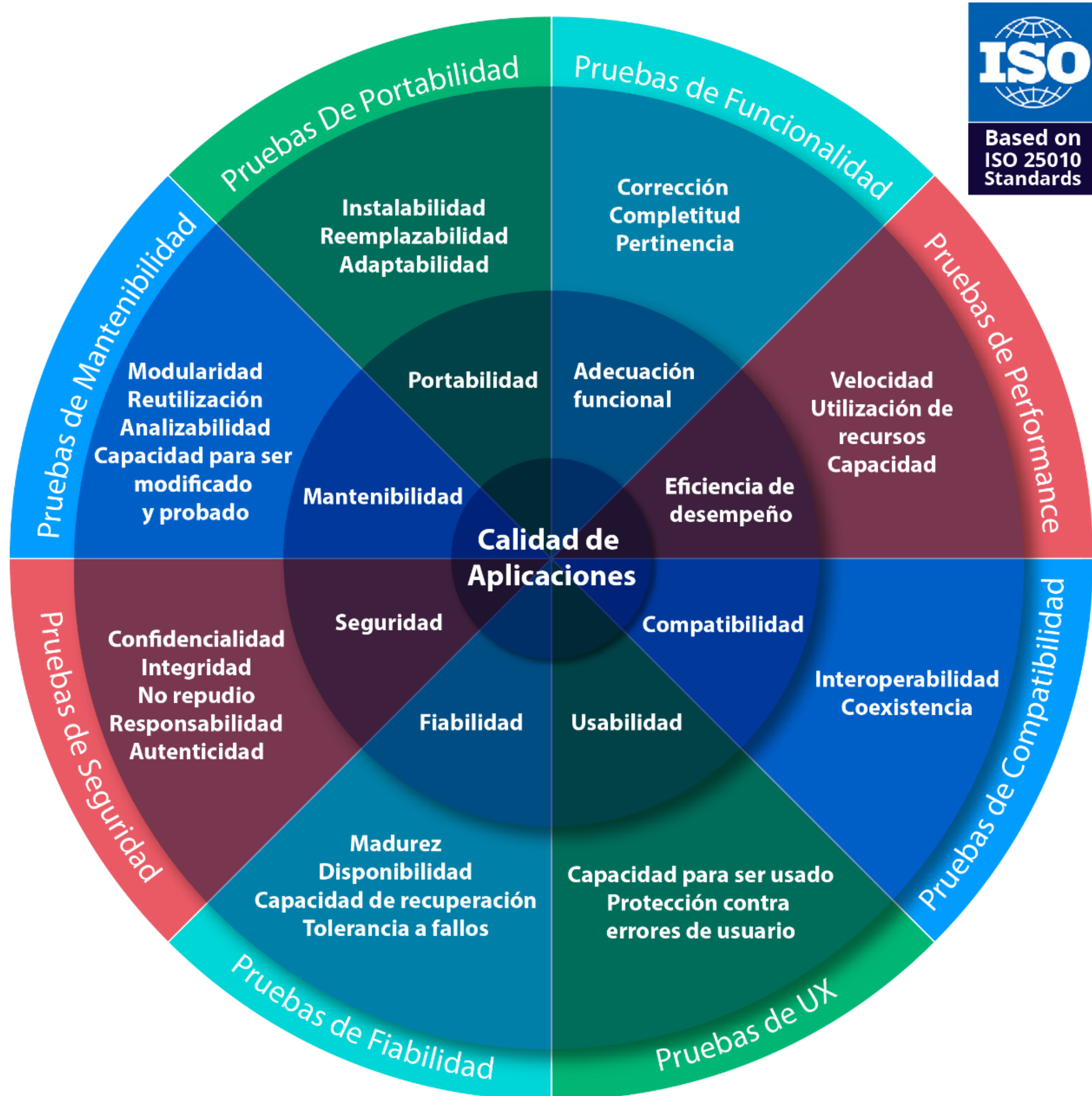
¿Se trata de encontrar la mayor cantidad de fallos posible? ¿Un tester que encuentra cien fallos en un día hizo mejor su trabajo que uno que apenas encontró 15? Si esto fuera así, la estrategia más efectiva sería centrarse en el módulo que esté "más verde", que tenga errores más fáciles de encontrar, 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 usuario más le van a molestar (molestar, entorpecer, agobiar, enloquecer, enfurecer, etc., todos esos sentimientos negativos que a veces nos generan las aplicaciones, y que por lo general fruncimos el ceño).

¡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.

[12] 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.

Tipos de Pruebas

Al hablar de calidad, en la sección anterior, decíamos que hay una lista de propiedades a evaluar en las aplicaciones de software. Ahora, ¿cuáles son esas propiedades? Veamos la siguiente imagen a la cual le llamamos "la rueda del testing".



Existe una norma ISO que plantea cuáles son los factores de calidad de un producto de software, la norma ISO 25.000[13]. En base a esa norma fue que armamos el diseño de la rueda anterior, mostrando cada uno de esos factores, y cómo para cada factor de calidad existe un tipo de pruebas. Es así como para "seguridad" existe el "testing de seguridad", para "usabilidad" existen las "pruebas de usabilidad", etc. En cierta forma, en la figura anterior queda mostrado que las actividades de testing dejan protegidas a cada una de las características de calidad que hay en el centro.

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

Se pueden distinguir dentro de estos atributos algunos que son de calidad interna y otros de calidad externa. Por ejemplo, la usabilidad sería externa, porque es algo que el usuario puede percibir fácilmente. Por otro lado, la mantenibilidad sería algo interno, ya que el usuario no lo puede percibir directamente. De todos modos, una salvedad que es importante destacar, si algo tiene poca mantenibilidad llega un momento en que ese factor interno se vuelve externo. Es decir, la mala mantenibilidad es notada por el usuario. Por ejemplo, cuando el usuario pide un cambio simple en el sistema y se tardan meses para poder resolver este cambio, ese factor interno termina impactando directamente al usuario.

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

[13] Norma ISO 25000 https://iso25000.com/index.php/en/

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. Por ejemplo, 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 como ingenieros aprendemos un proceso un poco estricto, que se refleja bastante en la industria[14]:

- Primero matemáticas y materias formativas que nos ayudan a crear el pensamiento abstracto y lógico.
- Programación básica, estructurada y usando constructores fundamentales.
- Programación con tipos abstractos de datos.
- Programación con algoritmos sobre estructuras de datos.
- Programación orientada a objetos (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[15], poniéndolo en práctica en un grupo de 12 a 14 personas. Recién en estas materias se ve *algo* de testing.
- Para los últimos años 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 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.

En los últimos años, relacionado a las metodologías ágiles, DevOps, *Continuous Integration y Continuous Delivery*, se ha estado hablando mucho de “*shift left testing*”. O sea, mover el testing a etapas más tempranas del desarrollo. Lo importante a destacar

[14] 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. Por suerte ha estado mejorando en los últimos años.

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

también, es que no hay que ver al ”testing” como una actividad pura y exclusiva de los testers. Hay muchas actividades de testing que deberían ser realizadas por otros integrantes del equipo, y en particular por los desarrolladores. Todos somos responsables de la calidad (no existe UN responsable de la calidad, ni un equipo responsable de la calidad, que solía ser el equipo de testers). Cada uno desde su rol es responsable que el resultado de todo el equipo sea un producto de buena calidad.

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.

Falacias sobre el Testing

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

- La falacia de la descomposición.
- La falacia de la composición.
- La falacia de que "todo testing es... testing".

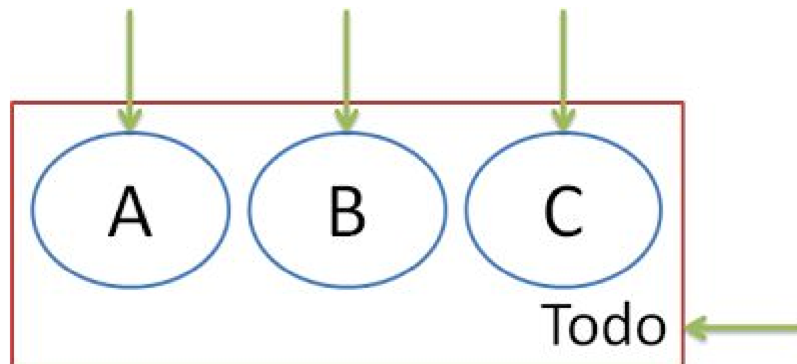
Primero, una falacia se define como un engaño, mentira o falsedad con lo que se intenta engañar a alguien. También, se pueden encontrar definiciones que dicen que es un razonamiento no válido o incorrecto, aunque con apariencia de razonamiento correcto pretende ser persuasivo.

En uno de mis libros preferidos relacionados al testing, *"Perfect Software: And Other Illusions about Testing"* de Jerry Weinberg[16], en uno de sus capítulos, pone nombres a varias experiencias y falacias que en más de una ocasión hemos vivido y volveremos a vivir profesionalmente.

En el libro Jerry cuenta una historia o anécdota para cada lección que nos quiere transmitir. En este caso cuenta sobre un joven, bien intencionado, aunque no experimentado gerente de proyecto. La mesa directiva y la alta gerencia ponía mucha presión sobre él y todos los equipos, observando muy de cerca la evolución de los proyectos. Había una apuesta muy fuerte, además de que ya los habían vendido a los clientes más importantes, por lo que era muy común que insistieran que el testing del sistema se hiciera sobre las pequeñas unidades del sistema. Haciendo esto se comprobaba que estaba 'funcionando', ergo, la implementación de la unidad tendría que posteriormente estar bien, y ahí nos encontramos con *la falacia de la descomposición*.

---

[16] Perfect Software: And Other Illusions about Testing
https://www.amazon.com/Perfect-Software-Other-Illusions-Testing/dp/0932633692



Falacia de Composición:

test(A) + test(B) + test(C)= test(Todo)

Falacia de Descomposición:

test(Todo) = test(A) + test(B) + test(C)

Una serie de reuniones en las que se habló sobre la locura que representaba solo probar las unidades sin considerar la integración entre las unidades y entre los sistemas (descomposición) llevó a intercambios de opiniones con el gerente de proyecto y a *la falacia de la composición*, donde se tiende a creer que si a un sistema lo probamos como un todo, entonces también probamos las partes, ¿verdad? En realidad, no. El que probemos un sistema como un todo, no asegura que probemos el correcto funcionamiento de las partes individuales y viceversa. Podemos haber cubierto toda la extensión de la aplicación, pero ¿cuánto podemos decir que cubrimos de los módulos o partes que la componen? ¿y su integración con el resto de los sistemas con los que tenía que interactuar? Podía pasar que una unidad dejara de funcionar con un impacto bajo o tener un impacto crítico dejando toda la aplicación sin funcionamiento, la falacia de la composición.

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

Testing Independiente[17]

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 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.

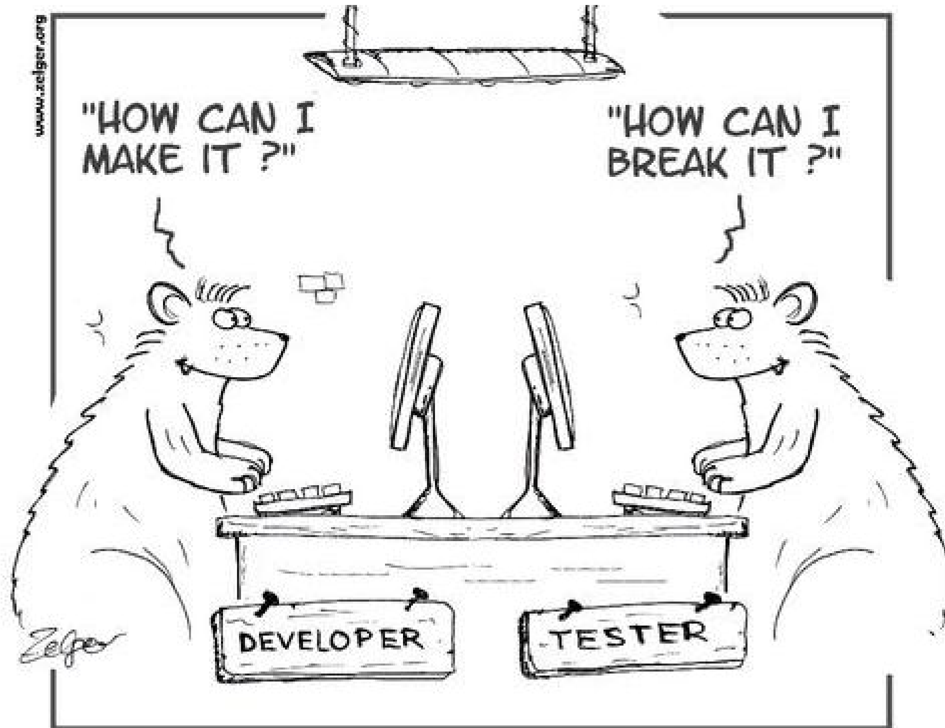
Tal como lo ha dicho alguna vez Antoine de Saint-Exupery, "*Es mucho más difícil juzgarse a sí mismo que juzgar a los demás*".

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 su calidad.

También hay otro tema de percepción, arrogancia (o no sé cómo llamarlo) que tiene que ver con que muchos desarrolladores ven al tester como el que sabe menos, que no sabe programar, que no participó en la construcción entonces no entiende. Imaginemos cómo podemos sentirnos de que esa persona nos marque errores...

La siguiente caricatura representa la idea que somos las mismas ratas de laboratorio, pero tal vez estamos enfrentados en cuanto a la forma de ver o de pensar sobre el mismo problema. Mientras el desarrollador se concentra en "cómo lo construyo" el tester piensa en "cómo lo destruyo".

[17] Basado en un fragmento de "Ingeniería de Software, un enfoque práctico" de Pressman.
https://www.goodreads.com/book/show/142783.Software_Engineering



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!

Estructura del Resto del Libro

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

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

- **Introducción a las Pruebas Funcionales**: comenzaremos por conceptos básicos e introductorios para seguir luego mostrando su utilidad, profundizando en un posible método básico para abordar la tarea de diseñar pruebas. Entre otras cosas veremos las técnicas que incluso se terminan utilizando casi en forma inconsciente, tales como: partición en clases de equivalencia, valores límites, etc.
- **Técnicas de Diseño de Pruebas Funcionales:** luego de haber visto las básicas de las pruebas funcionales podremos seguir describiendo algunas técnicas a las que llamaremos "avanzadas", siempre pensando en particular en sistemas de información. Entre ellas veremos cómo derivar casos de prueba a partir de casos de uso, tablas de decisión, máquinas de estado, etc.
- **Pruebas Exploratorias**: generalmente se las confunde con las pruebas ad-hoc[18], 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 trataremos el tema de 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.

[18] Ad-hoc: https://es.wikipedia.org/wiki/Ad_hoc

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.