Pruebas de software

Texto de apoyo de Algoritmos y Programación III

Facultad de Ingeniería de la Universidad de Buenos Aires

Carlos Fontela

Edición 2018

Prefacio

El presente trabajo es poco más que un apunte de clase, pensado para introducir el tema de pruebas de software a alumnos de una materia de programación avanzada[1].

Por lo tanto, no pretende cubrir todos los temas, sino sólo dar un pantallazo orientador, previo a materias de ingeniería de software en las cuales se retomarán muchos de los temas planteados aquí, aunque en mucha mayor profundidad[2].

En realidad, reconoce como antecedente un apunte que escribimos con Pablo Suárez hace ya mucho tiempo, y que había quedado muy obsoleto [Suárez 2003]. Viene básicamente a cubrir esa obsolescencia, que ya era poco menos que intolerable.

Al final figura una bibliografía y otras referencias para aquellos lectores que deseen ampliar lo aquí tratado. También, para facilitar la lectura, hay abundantes notas al pie de página que amplían lo dicho en el texto principal, pero que pueden ser omitidas en una lectura rápida.

Carlos Fontela

Marzo de 2018

[1] La materia en cuestión se llama Algoritmos y Programación III, y se dicta en las carreras de Ingeniería Informática, Licenciatura en Sistemas e Ingeniería Electrónica de la Universidad de Buenos Aires. Su objeto principal es la Programación Orientada a Objetos, más algunas cuestiones metodológicas y principios de diseño. Los alumnos vienen de otras dos materias de programación, una introductoria con el paradigma estructurado y una de estructuras de datos y algoritmia.

[2] Luego de esta materia, los alumnos de las carreras informáticas y de sistemas profundizan en temas de análisis, diseño, administración de proyectos, calidad, etc., dedicando a cada uno de estos temas una materia como mínimo.

Contenido

Contexto

La actividad de pruebas es una de las disciplinas características del desarrollo de software, tanto como la programación, el diseño o el análisis. En términos de la administración de proyectos, es parte del área de conocimiento de gestión de la calidad. Pero… ¿a qué llamamos calidad?

La calidad es una propiedad de cualquier producto que hace que quien lo usa se sienta conforme. Esta conformidad puede provenir de que el producto cumple lo que el usuario deseaba, las expectativas que tenía o simplemente lo ha impresionado en algún aspecto que hace que su percepción sea la de un producto de *buena calidad*.

Ahora bien, ¿cuándo se introduce la calidad en un producto? La respuesta es sencilla: cuando se lo construye. Por lo tanto, no son las pruebas las que confieren calidad al producto[3]. Las pruebas tienen solamente el objeto de detectar errores; y cuando los detecten, se deberá mejorar la calidad reparando el producto, lo cual siempre implica retrabajo.

Precisamente por eso, la manera de hacer más económico el desarrollo de cualquier producto, entre ellos el software, es trabajando mejor, de modo tal que las pruebas, que son una actividad de **control de la calidad (QC[4])**, sean (casi) siempre exitosas. Para ello, existen prácticas diversas denominadas de **aseguramiento de la calidad** (**QA[5]**), que buscan mejorar el proceso de desarrollo con vistas a que el producto final sea de calidad.

Como decía Edsger Dijkstra hace ya mucho tiempo, "las pruebas pueden ser una manera efectiva de mostrar la presencia de errores, pero no hay esperanza de que sean adecuadas para mostrar su ausencia"[6]. De manera que nunca vamos a encontrar todos los errores de un programa haciendo pruebas.

Este trabajo se centra en pruebas, y por lo tanto en QC. Sin embargo, a medida que avancemos veremos también algunas prácticas de QA.

Qué probamos

Verificación y validación

Cuando hablamos de pruebas, hay dos formas de verlas a la luz de lo que estén pretendiendo controlar: hay pruebas centradas en la verificación y otras centradas en la validación.

La **verificación** tiene que ver con controlar que hayamos construido el producto tal como pretendimos construirlo. La **validación**, en cambio, controla que hayamos construido el producto que nuestro cliente quería. La diferencia puede ser sutil, pero es importante.

Supongamos que un cliente nos pide que implementemos un juego de TaTeTi para dispositivos móviles. Luego de hablar con él un poco sobre colores, plataformas en las que se

---

[3] Como dice Steve McConnell, tratar de mejorar la calidad de un programa aumentando la frecuencia de las pruebas es como tratar de adelgazar pesándose más seguido [McConnell 2004]. Textualmente: "Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often."

[4] Acrónimo de Quality Control.

[5] Acrónimo de Quality Assurance.

[6] Textualmente: "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their abscense".

deba poder ejecutar, posibilidad o no de deshacer jugadas, etc., el equipo se da por satisfecho y se pone a trabajar. Diseña, codifica y prueba su programa. Al cabo de unos días, se reúne de nuevo con el cliente y le muestra el producto: impecable en diseño, se ejecuta sin problema en las dos plataformas pedidas, juega con la mejor de las tácticas contra el usuario… Sin embargo, el usuario afirma: "Esto no es lo que yo pedí: yo no quería jugar contra la aplicación, sino que permitiera el juego conectado de dos jugadores". Claramente, nuestro cliente no está satisfecho, pero ¿qué fue lo que falló?

Evidentemente, el usuario tenía una expectativa que no fue cubierta: el juego en red. También hay algo que el sistema le provee que él no deseaba y no está dispuesto a pagar: la inteligencia de la aplicación, que le permite jugar contra el usuario.

Ocurre que las pruebas que el equipo ejecutó estaban centradas en la verificación: entendimos algo, diseñamos una solución para el programa y verificamos que el programa hace lo que nos propusimos. Lo que falló fue la validación: lo que construimos no es lo que quiere el usuario.

Por supuesto, este tema da para mucho más en el contexto de la Ingeniería de Software, pero este es un pequeño texto de pruebas, así que no ahondaremos más…

Funcionalidades y atributos de calidad

Hay pruebas centradas en probar funcionalidades, cosas que el programa debe hacer. Estas pruebas surgen de modo natural de los requerimientos que el usuario expresa.

Por ejemplo, son enunciados de pruebas funcionales del juego de TaTeTi:

- Si intento colocar una ficha en una celda ocupada, el programa debe impedirlo.
- Si quiero jugar dos veces seguidas sin esperar que juegue el adversario, el programa debe impedirlo.
- Si coloco una cruz en un casillero, esa cruz debe permanecer allí hasta el final del juego.
- Si al colocar una cruz o círculo quedan tres cruces o círculos en línea, el usuario que jugó acaba de ganar y el juego debe terminar.

Decimos que las anteriores son **pruebas funcionales**.

Pero en ocasiones ocurre que debemos probar características del sistema que no son funcionales. Por ejemplo:

- El tiempo de respuesta del juego no puede ser superior a 1 segundo.
- El programa debe correr en las dos plataformas que soporten la mayor cantidad de dispositivos en uso en América Latina.
- Si se produce un error en el programa, el dispositivo móvil debe seguir funcionando para todas las demás aplicaciones.

Decimos que estas son **pruebas de atributos de calidad**[7].

Entre las pruebas de atributos de calidad, hay tantas clasificaciones como personas que se han dispuesto a formularlas. Aquí va un simple listado de algunos posibles tipos de pruebas de atributos de calidad:

[7] A veces llamadas "pruebas no funcionales".

- Pruebas de compatibilidad: chequean las diferentes configuraciones de hardware o de red y de plataformas de software que debe soportar el producto.
- Pruebas de rendimiento: evalúan el rendimiento del sistema (velocidad de respuesta, uso de memoria) en condiciones de uso habitual.
- Pruebas de resistencia o de estrés: comprueban el comportamiento del sistema ante situaciones donde se demanden cantidades extremas de recursos (por ejemplo: número de transacciones simultáneas anormal, excesivo uso de la memoria, redes sobrecargadas, etc.).
- Pruebas de seguridad: comprueban que sólo los usuarios autorizados puedan acceder a las funcionalidades que les corresponden y que el programa se mantenga estable ante intentos de vulnerar la seguridad del mismo.
- Pruebas de recuperación: chequean que el programa se recupere correctamente luego de una falla.
- Pruebas de instalación: verifican que el sistema puede ser instalado satisfactoriamente en el equipo del cliente, incluyendo todas las plataformas y configuraciones de hardware previstas.

Alcance de las pruebas

Alcance de las pruebas de verificación (o técnicas)

Las pruebas de verificación – esto es, pruebas que los propios desarrolladores ejecutan para ver que están logrando que el programa funcione como ellos pretenden – se pueden categorizar en pruebas unitarias o de integración.

Las **pruebas unitarias** verifican pequeñas porciones de código. En el paradigma de objetos, por ejemplo, verifican alguna responsabilidad única de un método, como una única postcondición. Se trata de pruebas que ejecutan los programadores para ver que lo que acaban de escribir hace lo que ellos pretendían. Luego de un lapso corto de programación, se debería ejecutar alguna prueba unitaria que compruebe que vamos por el camino correcto.

Las **pruebas de integración**, en cambio, prueban que varias porciones de código, trabajando en conjunto, hacen lo que pretendíamos. Por ejemplo, en el paradigma de objetos, se trata de pruebas que involucran varios métodos, o varias clases, o incluso subsistemas enteros.

Ambos tipos de pruebas son pasibles de ser automatizadas – esto es, escritas en código – mediante una enorme cantidad de herramientas disponibles para cada lenguaje y cada plataforma. Incluso se las puede integrar fácilmente en los entornos de desarrollo (IDE[8]). Las herramientas en cuestión suelen ser los frameworks[9] xUnit, llamados así genéricamente porque los primeros en surgir se denominaban *SUnit*, *JUnit*, *NUnit*, *unittest*, etc[10].

---

[8] Acrónimo de Integrated Development Environment.

[9] Un framework o marco de trabajo es una herramienta que realiza ciertas tareas basándose en código que le proveemos. Los frameworks de pruebas permiten definir código de inicialización, de prueba y de cierre, que luego el framework ejecuta en forma repetitiva y emite un informe.

[10] Más allá del "unit" que aparece en su nombre, suelen ser usados para todo tipo de pruebas técnicas, unitarias o de integración. Hay otros frameworks que se han cuidado de no usar la palabra "unit" como parte del nombre: *TestNG*, por ejemplo.

También, en ambos casos, las pruebas de verificación se pueden – y se suelen – escribir antes del código que prueban, aunque evidentemente se ejecuten después. Enfoques como los de TDD[11] refuerzan este proceder (ver TDD más adelante).

La ventaja de las pruebas unitarias está en que permiten aislarse del conjunto del sistema y analizar que una pequeña porción de código se comporta como se espera. Hay ocasiones en que no es sencillo hacer una prueba unitaria, porque estamos intentando probar código que necesita de otros objetos, métodos o funciones para poder funcionar: en esas situaciones se utilizan objetos ficticios[12], que ayudan a aislar el código a probar.

Las pruebas de integración no siempre prueban el sistema como un todo. A veces prueban también partes del mismo que se quieren aislar de otras, sea porque aún no han sido construidas, sea porque se quiere evitar accesos a redes o bases de datos que harían muy lento el desempeño de las pruebas, sea porque se quiere aislar el comportamiento del sistema de la interacción con el usuario, etc. En estos casos, se pueden utilizar las mismas técnicas y herramientas de objetos ficticios que se usan para aislar las pruebas unitarias.

Las pruebas de verificación podrían ser de caja negra o de caja blanca.

Decimos que una prueba es de **caja negra** cuando la ejecutamos sin mirar el código que estamos probando. Se trata más bien de una prueba ciega, que toma el código a probar como algo cerrado, que ante ciertos estímulos realiza determinadas acciones. Cuando, en cambio, analizamos el código durante la prueba, decimos que es una prueba de **caja blanca**. En general, se prefiere hacer pruebas de caja negra, precisamente porque se desea probar el funcionamiento y no verificar la calidad del código. Se recurre a la técnica de caja blanca cuando algún error se resiste a ser encontrado y debemos examinar el código en detalle. Por su propia definición, el *debugging*[13] es una técnica de caja blanca.

Una prueba de verificación de caja blanca que ha caído en desuso es la llamada **prueba de escritorio**. En ella, el programador da valores a ciertas variables y recorre mentalmente el código, ayudándose con anotaciones en papel, para ver que se comporta como él espera. El problema de las pruebas de escritorio es que, al ser el mismo programador el que revisa lo que él escribió, puede estar sesgado por sus propios errores. Además, la prueba de escritorio simula lo que haría la computadora, lo cual tenía sentido cuando el tiempo de computación era muy caro. Hoy en día, sin embargo, es más fácil, económico y seguro hacer que trabaje la propia computadora[14].

Otra técnica interesante es la verificación de la calidad del código mediante **revisiones de código** realizadas de a dos o más programadores. Si bien se practica poco en la forma tradicional, es bastante usual hacerla con la ayuda de herramientas, y muy especialmente en el desarrollo de software libre[15]. Suele ser útil para que surjan algunos problemas que al programador le hayan pasado desapercibidos, en parte por los comentarios de los pares y en

---

[11] Test-Driven Development o "desarrollo guiado por las pruebas".

[12] Un **objeto ficticio** o **doble de prueba** es un objeto que se crea especialmente para simular la existencia de comportamiento que aún no ha sido implementado. Hay muchos tipos de objetos ficticios: Stubs, Mocks, Fakes, etc. Un análisis detallado puede verse en el excelente libro de Gerard Meszaros [Meszaros 2007].

[13] Debugging, a veces traducido al castellano como "depuración", es el proceso de identificar y corregir errores de programación.

[14] Las herramientas de debugging de los entornos de desarrollo permiten hacer exactamente esto.

[15] En el desarrollo de software libre, estas revisiones se suelen hacer en forma diferida y remota.

parte por el solo hecho de revisar. Precisamente, la práctica de programar de a pares[16] que propone Extreme Programming[17] se enfoca en una revisión constante de código.

Alcance de las pruebas de validación (o de usuarios)

Las pruebas de validación, por sus propios objetivos, parecería que las deben ejecutar los usuarios. Podría ser… pero hay alternativas.

En general, se suele trabajar con pruebas de aceptación diseñadas por usuarios – o al menos en conjunto con usuarios, o en el caso extremo, diseñadas por el equipo de desarrollo y validadas por usuarios – pero que ejecuta el equipo de desarrollo. Estas pruebas, se suelen denominar **pruebas de aceptación de usuarios** (**UAT**[18]). Para que sean auténticas UAT se deberían ejecutar en un entorno lo más parecido posible al que va a utilizar el usuario.

No obstante, cuando un sistema debe salir a producción y tiene cierta criticidad, se suele poner a disposición de usuarios reales para que ellos mismos ejecuten las pruebas. Si las pruebas las hacemos en un entorno controlado por el equipo de desarrollo, las llamamos **pruebas alfa**. Si, en cambio, el producto se deja a disposición del cliente para que lo pruebe en su entorno, las llamamos **pruebas beta**.

Todas estas pruebas que hemos mencionado suponen que ejecutamos el sistema completo, con sus conexiones con otros sistemas, con su interfaz de usuario, sus medios de almacenamiento de datos persistentes, etc. Hay ocasiones en que deseamos probar solamente comportamiento, en el sentido de que la lógica de la aplicación es correcta, pero sin interfaz de usuario. A estas pruebas más limitadas las denominamos **pruebas de comportamiento**[19].

La razón que se suele esgrimir para diseñar y ejecutar pruebas de comportamiento, es que se centran en el comportamiento del sistema, más estable que las cuestiones de diseño de la interfaz de usuario. De todas maneras, no hay que olvidar que estamos hablando de pruebas centradas en la validación, así que la cooperación de los usuarios es tan importante como en las pruebas de aceptación. Aclaramos esto porque esta cooperación, que necesitamos realmente, puede ser puesta en riesgo porque los usuarios no comprendan el comportamiento del sistema al no verlo a través de una interfaz que les resulte familiar.

Hay muchas herramientas para automatizar pruebas de comportamiento, con formatos muy diferentes, pero que en todos los casos pretenden resultar amigables para los usuarios no técnicos. Hay herramientas basadas en tablas, como *FIT* y *FitNesse*, otras basadas en texto con formato especial, como *RSpec*, *JBehave*, *Cucumber* y *Behave*, y las hay también de texto más libre, como *Concordion*.

En cuanto a las herramientas para realizar UAT a través de la interfaz de usuario, las herramientas dependen mucho de la plataforma de ejecución del programa: si es una aplicación de escritorio, móvil, web, etc. Por ejemplo, para aplicaciones web existen herramientas que graban la ejecución y permiten reproducirla luego simulando a un usuario

---

[16] La programación en pareja (*pair programming* en inglés) requiere que dos programadores participen en un esfuerzo combinado de desarrollo en un mismo sitio de trabajo.

[17] La programación extrema o Extreme Programming (XP) es un método ágil de desarrollo de software basado en prácticas reunidas por Kent Beck [Beck 1999].

[18] Acrónimo de User Acceptance Test.

[19] Fowler las llama *pruebas subcutáneas*, porque operan apenas debajo de la interfaz de usuario [Fowler 2011].

humano[20], como *Selenium IDE*, y otras más sofisticadas, que permiten programar esa interacción con algunas lenguajes de programación, como *Selenium Webdriver*.

De todas maneras, las UAT que se suelen automatizar son las funcionales y algunas que prueban atributos de calidad. Otros atributos de calidad, como los que implican experiencia del usuario (la usabilidad, la navegabilidad y la estética, por ejemplo), no se pueden probar sino manualmente.

Pruebas en producción

Por supuesto, se puede también poner el sistema en producción y esperar a ver qué errores surgen o qué problemas encuentran los usuarios. A esto lo llamamos realizar las pruebas en producción.

A priori, puede parecer un enfoque deshonesto: estaríamos entregando un producto sin probar para que sean los propios usuarios los que enfrenten los problemas. Además, puede ser muy costoso, ya que deberíamos arreglar los problemas en forma tardía.

Sin embargo, en las aplicaciones en que los errores no causen grandes riesgos[21], podría ser una opción. Máxime en los casos de sistemas distribuidos, que interactúan de maneras difíciles de predecir y son muy costosos – y a veces imposibles – de probar en forma exhaustiva. En estos casos, se pueden realizar pruebas en un ambiente pre-productivo para los escenarios más críticos, y terminar de probar en producción.

Hay varios tipos de pruebas que se realizan en producción: monitoreo del uso, pruebas exploratorias, pruebas A/B[22], etc.

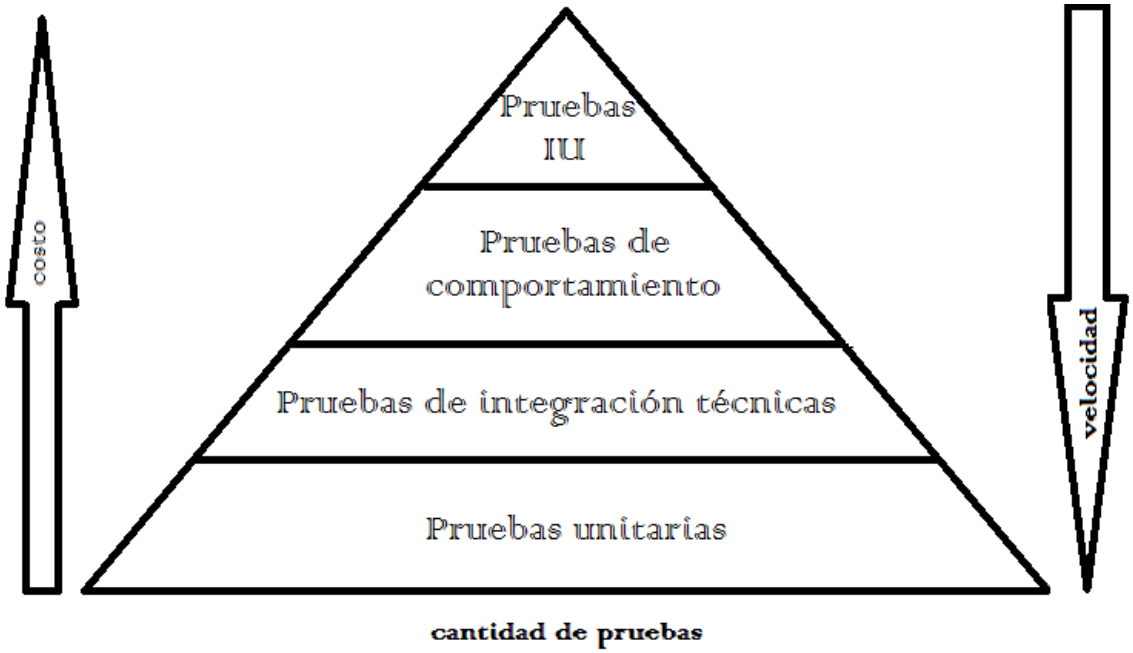
La pirámide de pruebas y sus razones

Veamos esta figura, que con algunas variantes puede encontrarse en toda la web:

[20] Se les llama generalmente herramientas de tipo "Record and Replay" y generan guiones editables a partir de la primera ejecución de una prueba.

[21] Claramente, si un error pone en riesgo vidas humanas o hace perder importantes cantidades de dinero, esta opción debería ser descartada como norma.

[22] Las pruebas A/B son un tipo de pruebas beta que se usan habitualmente para ver cómo responden distintos grupos de usuarios a una variable binaria, a modo de experimento, antes de habilitar una característica para todos ellos. Si bien es un mecanismo típico de las pruebas de usabilidad, se pueden usar para pruebas de comportamiento también, para detectar problemas en producción.



Lo que indica la figura es que hay distintos niveles de prueba, pero que las que más deben abundar son las unitarias y las que menos las de aceptación a través de la interfaz de usuario.

La razón de ser de esta pirámide es más bien económica:

- Las pruebas unitarias se ejecutan más rápidamente que las pruebas de integración, éstas más que las de comportamiento, las cuales a su vez son más veloces que las de aceptación. Por lo tanto, cuanto más abajo en la pirámide, más rápido se ejecutan, con un doble provecho: se pierde menos tiempo esperando y se logra mantener la concentración de los desarrolladores, que mantienen el foco en lo que están haciendo.
- Las pruebas de aceptación son más frágiles que las de comportamiento, pues dependen de la interfaz de usuario, que suele ser cambiante; y a mayor fragilidad, mayor tiempo gastado en mantenimiento de las pruebas. A su vez, a medida que bajamos en la pirámide, la fragilidad disminuye debido a la menor dependencia de otras partes del sistema[23].

No obstante, en todo sistema bien construido, debería haber pruebas de todos los niveles.

Un corolario interesante es que, si una prueba a través de la interfaz de usuario falla, se debe a una de dos posibilidades: o es una funcionalidad que no fue correctamente probada en las pruebas de comportamiento o el problema está en la propia interfaz de usuario. Si se debe a lo primero, deberíamos ver cómo introducir una prueba de comportamiento que evidencie el problema. Lo mismo ocurre con cada nivel: si falla una prueba de integración, puede que sea realmente un problema de integración o que debamos ver si no nos está faltando alguna prueba unitaria que evidencie esa falla.

[23] Sin embargo, hay que tener cuidado con las generalizaciones. Si bien se considera a las pruebas unitarias como muy estables, lo cierto es que, al ser representativas de la implementación detallada, pueden cambiar sin que cambie la funcionalidad.

Roles del desarrollo ante las pruebas

Visiones tradicionales

Dentro de la organización de desarrollo, la visión tradicional sostiene que debe haber personas con el rol de programadores y otras con el rol de testers[24]. Los testers son quienes ejecutan las pruebas y velan porque el producto llegue sin errores a los clientes y usuarios. Por lo tanto, deben ejecutar pruebas de aceptación en un ambiente especial y sólo a ellos les corresponde autorizar la liberación del producto para su entrega.

Por supuesto, la visión tradicional no ignora que las pruebas no agregan calidad, sino que sólo la controlan. La calidad es introducida por los programadores, quienes en primera instancia son los responsables de que el programa no tenga errores. En efecto, el caso ideal sería que los testers reciban un producto sin errores y se limiten a chequear que no los haya realmente.

De esta manera, la visión tradicional separa roles y responsabilidades en dos equipos distintos dentro del desarrollo: el equipo de programadores y el equipo de testers. Se basa en la noción administrativa de control cruzado por contraposición de intereses, que hace que el que realiza un trabajo no puede ser el mismo que lo controla. Sin embargo, en el desarrollo de software es una noción cuestionada desde hace un par de décadas.

¿Y los usuarios no prueban? En principio, en este trabajo estamos abordando las pruebas que se ejecutan en el marco del desarrollo. Sin embargo, es esperable que – salvo en programas de juguete – los usuarios prueben los programas que van a usar.

Ya hablamos de las pruebas alfa y beta. Precisamente, las organizaciones que adquieren un producto de software deben tener un ambiente de pruebas, de modo tal de probar el sistema que van a utilizar. Recién después de estas pruebas – que serían beta según nuestra clasificación – se debería pasar al ambiente productivo.

La visión ágil

A principios de siglo, cuando surgieron los métodos ágiles de desarrollo de software, se puso en cuestión la noción de control cruzado con contraposición de intereses, como no aplicable a la industria del software.

Es así que varios métodos ágiles, entre ellos Extreme Programming y Scrum[25], plantearon que todo el equipo de desarrollo debe trabajar de consuno y en aras de entregar un producto de calidad. Por lo tanto, la responsabilidad por entregar un producto de calidad y sin errores pasó a ser de todo el equipo de desarrollo (programadores y testers).

Scrum incluso llega a plantear que no debe haber roles diferentes dentro del equipo: todos son parte del Scrum Delivery Team, y no hay distinción desde afuera – ni siquiera para el Scrum Master[26] – entre programadores y testers. El equipo puede organizarse, si así lo desea, en programadores y testers, pero es una prerrogativa del equipo de desarrollo y no de nadie de

[24] El término inglés “tester” a veces figura en la bibliografía castellana como “probador”. Como no es un término usado en la República Argentina, donde resido, ni lo he escuchado en ningún otro lugar de habla castellana, mantengo el término más habitual.

[25] Scrum es un método ágil de desarrollo de software presentado por Ken Schwaber [Schwaber 1995].

[26] El rol de Scrum Master, o Facilitador como se lo llama a veces, es un rol de Scrum cuyo trabajo primario es eliminar los obstáculos que impiden que el equipo alcance sus objetivos, cuidando que el proceso se usa correctamente. No debe confundirse con un líder de equipo, ya que en Scrum los equipos son auto-organizados.

afuera del mismo. Vistos desde afuera, todos son *desarrolladores*, sin distinción entre programadores y testers. Así, la responsabilidad por la calidad del producto entregado no es de un rol en particular sino del equipo como un todo.

Lo que sí ocurre en la vida real es que algunos equipos separan los roles de programador y tester, pero trabajan en el mismo equipo, en el mismo lugar, y no hay personas compartidas con otros equipos de trabajo ni tienen autoridad por fuera del equipo. En otros equipos, todos los desarrolladores son programadores y testers a la vez, y se rotan para hacer las pruebas. Lo que se destaca también en los métodos ágiles es que, aun cuando haya roles diferentes de programadores y testers, deben trabajar juntos, al menos para que el programador pueda comprender la perspectiva del tester, aprenda a programar código más sencillo de probar y el tester pueda aprender a automatizar pruebas. Pero insistamos en un concepto clave: esta es materia que debe definir el equipo hacia adentro, no algo que admita injerencia externa.

Pruebas automatizadas: quién las desarrolla

Como ya dijimos, hay gran cantidad de tipos de pruebas que pueden automatizarse, en el sentido de que se pueden escribir en código y luego ser ejecutadas por una computadora. Ahora bien, ¿quién debe codificar las pruebas automatizadas?

Las pruebas unitarias son pruebas de programador, así que sin duda las deben escribir los programadores y ejecutarlas ellos mismos o un proceso automatizado de integración (ver más adelante integración continua, donde se trata este tema).

Las pruebas de integración técnicas también son pruebas de programador: conviene que las escriban los programadores y que las ejecuten sobre servidores de integración.

Las pruebas de comportamiento están a mitad de camino entre pruebas de programadores y pruebas de testers. Por lo tanto, cualquiera de los dos roles podría desarrollarlas, en el caso de que existan estos roles diferenciados: si no, son pruebas ideales para roles mixtos. En algunos enfoques metodológicos como BDD, SBE o ATDD (ver más adelante) se espera que estas pruebas salgan de talleres multidisciplinarios que incluyan programadores, testers y usuarios, así que se presume que la construcción de estas pruebas es colaborativa y los tres roles son autores de las mismas.

Con las UAT automatizadas pasa algo parecido, pero desde el otro extremo. En principio, se pretende que las escriban usuarios o analistas de negocio. Si esto no se consiguiera, las podrían escribir los testers (o los desarrolladores, si el rol de tester no estuviera separado), pero es imperativo validarlas con los usuarios para no construir algo diferente a lo que ellos desean. Finalmente, si estamos trabajando en el marco de BDD, SBE o ATDD, surgirán de los talleres multidisciplinarios.

Pruebas manuales: quién diseña la prueba

Vimos que, si bien podemos automatizar varios tipos de pruebas, hay ocasiones en que éstas sólo pueden ser ejecutadas por humanos.

Hoy en día se suelen dejar como pruebas manuales algunos tipos de UAT que luego ejecutarán testers (en el ambiente de pruebas de la organización de desarrollo) y clientes (en general, en el ambiente de pruebas del cliente). Estas pruebas no son pruebas de programadores y no deberían escribirlas ellos mismos. En la visión tradicional podrían ser desarrolladas por analistas o los mismos testers. En la visión ágil, dado que programadores y testers no son roles disjuntos, habría que dar la mayor participación posible a los usuarios.

Pruebas y desarrollo

Como dijimos al comienzo, si bien este es un libro de pruebas, y éstas son predominantemente una técnica de control de calidad (QC), existen muchas prácticas metodológicas de aseguramiento de la calidad (QA) que se basan en pruebas. Es el momento de hacerles justicia haciendo un breve recorrido por las mismas.

Antes de seguir, una aclaración: se ha puesto de moda denominar *desarrollo* a la programación y *desarrolladores* a los programadores[27]. Entiendo que es un error, porque de esa manera confundimos todo. El desarrollo de software es una disciplina que involucra actividades y especialidades diferentes: análisis, diseño, programación, pruebas, despliegue, etc. Por lo tanto, las pruebas son una actividad más del proceso de desarrollo, y así lo entenderemos en este trabajo.

Cuándo probamos

El primer ciclo de vida del desarrollo de software suponía que las pruebas se ejecutaban íntegramente como última etapa del desarrollo de un proyecto, justo antes de la puesta en producción. En la práctica, esto nunca funcionó del todo así: no tenía sentido esperar tanto, ya que, como a medida que se terminaba de desarrollar una funcionalidad, se la podía probar, se adelantaba trabajo probando antes[28].

En los años 1990, con el auge de los procesos iterativos, se incorporó la idea de probar al final de cada iteración. En los años 2000, los métodos ágiles, abogaron por la prueba más o menos continua, automatizando todo lo posible y haciendo que las pruebas guiaran el proceso de desarrollo, como veremos un poco más adelante.

Hoy en día, con el surgimiento de ciclos de flujo continuo, como el de entrega continua que veremos luego, las pruebas son continuas a lo largo de todo el desarrollo.

Todo lo que dijimos anteriormente se refiere a las pruebas sobre nuevas funcionalidades. Por supuesto, cada vez que se incorporan características a un programa, podemos provocar que dejen de funcionar cosas que ya habían sido probadas y entregadas: esto es lo que se llama una **regresión**. Para evitarlas, se ejecutan cada tanto, **pruebas de regresión**, que no es otra cosa que ejecutar las pruebas de todo el sistema a intervalos regulares: por supuesto que la automatización ayuda mucho a hacer estas pruebas de regresión más llevaderas.

Ventajas de la automatización

Dijimos ya varias veces que algunas pruebas se pueden automatizar. Entendemos por automatización al hecho de que las pruebas del programa se desarrollen en código, de forma tal que, con la sola corrida del código de pruebas, deberíamos saber si lo que estamos probando funciona bien o mal.

Pero, ¿qué ventajas tiene automatizar?

Las ventajas más obvias son las siguientes:

- Nos independizamos del factor humano, con su carga de subjetividad y variabilidad en

[27] No sé si es algo generalizado, pero en la Argentina es bastante habitual desde hace unos años.

[28] No obstante, había quienes criticaban este enfoque, afirmando que al adelantar pruebas se perdía tiempo, ya que de todas maneras habría que hacer pruebas de integración al final.

el tiempo.

- Es más fácil repetir las mismas pruebas, con un costo ínfimo comparado con las pruebas realizadas por una persona. Esto es aplicable a regresiones, *debugging* y errores provenientes del sistema ya en producción.
- Las pruebas en código, escritas con las herramientas adecuadas, sirven como herramienta de comunicación, minimizando las ambigüedades.

Hay un conjunto de prácticas que se fueron desarrollando alrededor de las pruebas automatizadas, y que analizaremos en los próximos ítems.

Tipos de pruebas y automatización

<table>
<tr><td></td><td colspan="2">Tipo de comportamiento</td></tr>
<tr><td></td><td>Por funcionalidad</td><td>Atributos de calidad</td></tr>
<tr><td rowspan="3">Orientadas al negocio

Orientadas a la tecnología</td><td>Pruebas funcionales de aceptación
Pregunta: ¿funciona como el usuario desea?
Automatizables con varias herramientas</td><td>Pruebas de experiencia de usuario
Pregunta: ¿es sencillo de usar y de aprender?
Pruebas manuales solamente</td></tr>
<tr><td>Pruebas de integración técnicas
Pregunta: ¿anda bien el sistema como un todo?
Automatizables con herramientas xUnit</td><td>Pruebas exploratorias
Pregunta: ¿es consistente y agradable?
Pruebas manuales solamente</td></tr>
<tr><td>Pruebas unitarias
Pregunta: ¿está bien programado el código?
Automatizables con herramientas xUnit</td><td>Pruebas de otros atributos de calidad
Pregunta: ¿es efectivo, seguro, escalable, etc.?
Automatizables con herramientas especiales</td></tr>
<tr><td></td><td>Soporte para el desarrollo</td><td>Criticidad del producto</td></tr>
<tr><td></td><td colspan="2">Propósito de las pruebas</td></tr>
</table>

Ya hemos venido mostrando qué tipos de prueba podemos automatizar y cuáles no. Para aclarar un poco más, presentamos el diagrama de más arriba, que he tomado de [Meszaros 2007] y adaptado[29].

TDD

TDD fue la primera práctica basada en automatización de pruebas que estuvo bien soportada por herramientas. La presentó Kent Beck en el marco de Extreme Programming y enseguida surgieron frameworks para facilitar ser llevada a la práctica.

Básicamente, incluye tres sub-prácticas:

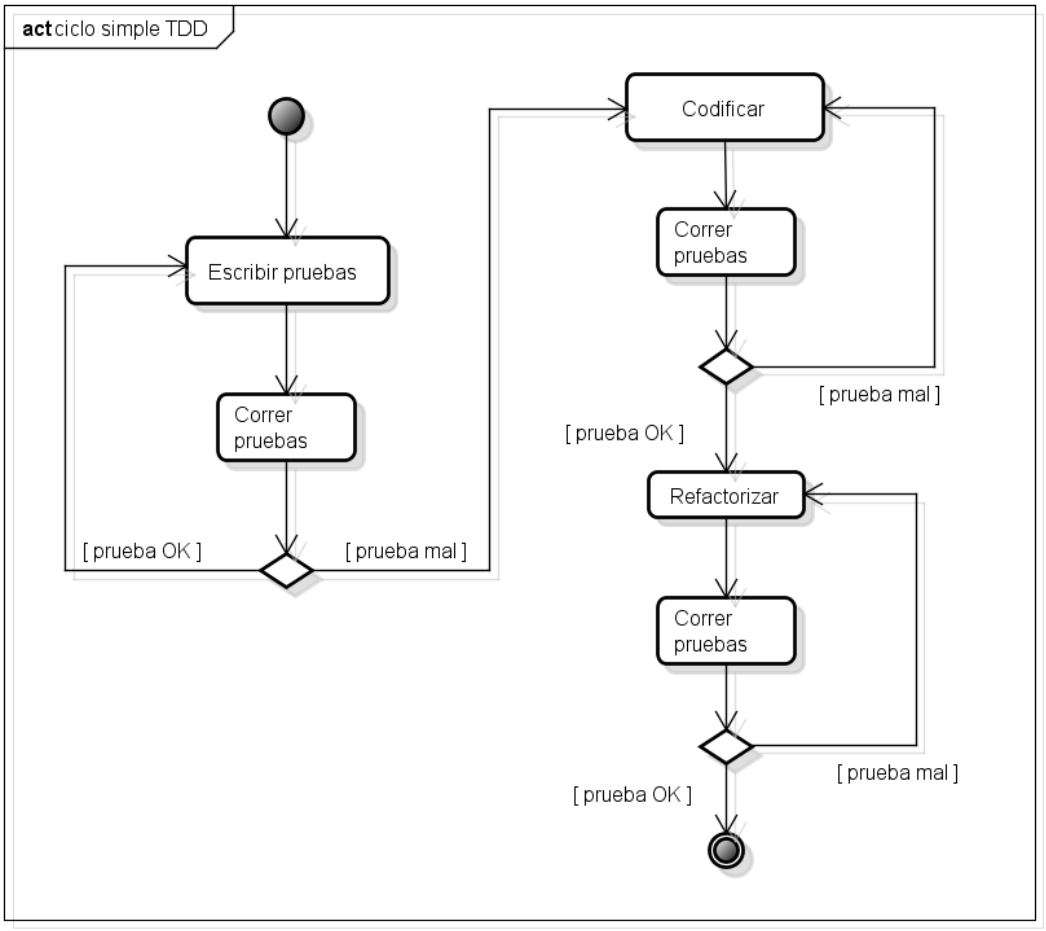
- Automatización: las pruebas del programa deben ser hechas en código, y con la sola ejecución del código de pruebas debemos saber si lo que estamos probando funciona bien o mal.
- Test-First: las pruebas se escriben antes del propio código a probar.

[29] Meszaros utiliza un cuadro parecido, y dice que surge a su vez de trabajos previos de Mary Poppendieck y Brian Marick [Meszaros 2007].

- Refactorización posterior: para mantener la calidad del código, se lo cambia sin cambiar la funcionalidad, manteniendo las pruebas como reaseguro.

Si bien no está dicho de forma clara, TDD – al menos como se lo usa hoy en día – denomina "pruebas" a las pruebas unitarias.

El siguiente diagrama de actividades muestra el ciclo de TDD[30]:



Las ventajas de TDD son:

- Las pruebas en código sirven como documentación del uso esperado de lo que se está probando, sin ambigüedades.
- Las pruebas escritas con anterioridad ayudan a entender mejor lo que se está por desarrollar.
- Las pruebas escritas con anterioridad suelen incluir más casos de pruebas negativas que las que escribimos a posteriori.
- Escribir las pruebas antes del código a probar minimiza el condicionamiento del autor por lo ya construido. También da más confianza al programador sobre el hecho de que el código que escribe siempre funciona.

[30] En los tres casos en que el diagrama dice "correr pruebas" se refiere a todas las pruebas generadas hasta el momento. Así, el conjunto de pruebas crece en forma incremental y sirve como pruebas de regresión ante agregados futuros.

- Escribir las pruebas antes del código a probar permite especificar el comportamiento sin restringirse a una única implementación.
- La automatización permite independizarse del factor humano y facilita la repetición de las mismas pruebas a un costo menor.
- La refactorización constante facilita el mantenimiento de un buen diseño a pesar de los cambios que, en caso de no hacerla, lo degradarían.

Pruebas orientadas al cliente

Pero no todo son pruebas unitarias, ni todo son pruebas de programador.

Cuando Kent Beck presentó Extreme Programming hizo especial hincapié en especificar mediante historias de usuario[31] acompañadas por pruebas de cliente [Beck 1999]. Está bastante claro en sus primeras publicaciones que Beck se refiere a pruebas de comportamiento. Pero con el tiempo, la práctica fue derivando en lo que se conoció como TDD, basado en pruebas unitarias que sirven como especificación del comportamiento de pequeñas porciones de código.

A mi juicio, esta transición ha sido desafortunada, ya que – si bien es una buena práctica guiar el diseño mediante pruebas unitarias – ha minimizado la importancia de las pruebas de aceptación para derivar el comportamiento del sistema.

Con el tiempo, fueron surgiendo algunas prácticas que buscaron subsanar este problema. He analizado en detalle estas prácticas, sus similitudes y diferencias en un trabajo de hace unos años [Fontela 2012]. Lo que sigue es una simple enumeración:

- BDD (Behavior Driven Development) [North 2006]: inicialmente surgió como una práctica para hacer bien TDD y fue convergiendo a una manera de especificar el comportamiento esperado mediante escenarios concretos que se puedan automatizar como pruebas de aceptación.
- STDD (Storytest Driven Development) [Mugridge 2008]: es una práctica que pretende construir el software basándose en ir haciendo pasar las pruebas de aceptación – que se pretenden automatizadas – que acompañan las historias de usuario.
- ATDD (Acceptance Test Driven Development) [Koskela 2007, Gärtner 2012]: similar a la anterior, construye el producto en base a pruebas de aceptación de usuarios, con menos énfasis en la automatización de las pruebas y más en el proceso en sí.
- SBE (Specification By Example) [Adzic 2009, 2011]: presentada originalmente como una práctica para mejorar la comunicación entre los distintos roles de un proyecto de desarrollo, ha ido convirtiéndose en una práctica colaborativa de construcción basada en especificaciones mediante ejemplos que sirven como pruebas de aceptación.

Desde el punto de vista del uso, podemos considerar a las cuatro prácticas como sinónimos, sobre todo porque en todos los casos se parte de ejemplos que sirven tanto como especificaciones del usuario, guías para el desarrollo y casos de aceptación a probar.

Estas técnicas usan un enfoque de afuera hacia adentro, partiendo de funcionalidades y sus casos de aceptación para luego ir refinando el diseño. En definitiva, por cada prueba de aceptación que debemos encarar, hay que hacer varios ciclos de desarrollo basados en pruebas unitarias y de integración técnicas.

---

[31] Historia de usuario o *user story* es una representación de un requisito escrito en una o dos frases utilizando el lenguaje común del usuario y que sirve como parte de la especificación y, en conjunto, para administrar el alcance del proyecto.

Por supuesto, al ir avanzando el desarrollo, las pruebas técnicas van a terminar cubriendo el mismo código que las pruebas de aceptación que las motivaron. De todas maneras, esto dista de ser un problema, pues la coexistencia de pruebas a distintos niveles va a facilitar el mantenimiento del sistema, sea cuando hay que realizar cambios de comportamiento o cuando haya que hacer grandes refactorizaciones [Fontela 2013]. Por eso, es una buena idea mantener todas las pruebas, aun cuando haya redundancia en la cobertura: a lo sumo, si muchas pruebas implican mucho tiempo de ejecución de las mismas, se puede no ejecutar todas las pruebas en todas las integraciones, siguiendo las ideas de Test Impact Analysis [Hammant 2017].

Integración y entrega continuas

Otra práctica que surgió a partir de las recomendaciones de Kent Beck fue la de **integración continua (CI[32])**. La idea es facilitar y automatizar al máximo las integraciones antes de liberar el producto que se está desarrollando. Básicamente consiste en realizar la compilación, construcción y pruebas del producto en forma sucesiva y automática como parte de la integración. Las integraciones deben terminar siempre en el tronco principal de la herramienta de control de versiones[33] y – al haber sido hechas luego de ejecutar las pruebas – se presume que están libres de errores. Por supuesto, si hubiera un error en las pruebas manuales posteriores o en el sistema en producción, se lo debe corregir de inmediato.

Como una derivación de CI surgió la práctica de **entrega continua (CD[34])**. Mientras que en CI la línea de automatización termina con una integración en el ambiente de desarrollo, CD pretende que el código siempre esté en condiciones de ser desplegado en el ambiente productivo. Por lo tanto, debemos incluir todas las pruebas de aceptación que se puedan automatizar. Como las pruebas de aceptación suelen ser más lentas de ejecutar que las pruebas unitarias, se puede ser más laxo en cuanto a la frecuencia de las ejecuciones de las mismas, pero en ese caso deben hacerse al menos una vez al día, en conjunto con las pruebas manuales que no se puedan automatizar.

Finalmente, hay una técnica denominada **despliegue continuo[35]** que, no sólo pretende que se esté permanentemente en condiciones de desplegar, sino que efectivamente cada pequeño cambio se despliegue en el ambiente de producción. Debido a la necesidad de desplegar en forma tan frecuente, suele descansar en ejecuciones sistemáticas de pruebas en producción.

Con estas prácticas se disminuye el riesgo de la aparición de errores en las pruebas, porque cualquier problema que surja es atribuible al último tramo de código desarrollado e integrado.

¿Cómo se diseña una prueba?

Más allá de toda la cuestión teórica de pruebas que venimos analizando, ¿cómo creamos una prueba cuando debemos hacerlo?

Diseño de pruebas unitarias y técnicas en general

Empecemos por las pruebas unitarias. Si pudiésemos hacer pruebas de caja blanca, podríamos

---

[32] Acrónimo de Continuous Integration.

[33] Esto no es aplicable a proyectos de código abierto, en los que cada contribuyente trabaja sobre una rama separada por períodos largos.

[34] Acrónimo de Continuous Delivery.

[35] "Continuous Deployment" en inglés.

tratar de comprender el código y asegurarnos de que cada rama de los condicionales y ciclos queden cubiertas. Sin embargo, resistamos la tentación de las pruebas de caja blanca. En efecto, si estamos tratando de probar el funcionamiento de un módulo cualquiera, no deberíamos escribir pruebas pensando en una implementación específica. Más aún, si estamos usando TDD, es seguro que el código ni siquiera existe cuando escribimos la prueba, así que mal podríamos usar una técnica de caja blanca. Entonces, analicemos la construcción de pruebas de caja negra.

Una primera forma de probar es ponernos en el rol de quien usa el módulo a probar, que en el caso de las pruebas unitarias es el mismo programador. Lo que podríamos hacer es seguir las nociones del diseño por contrato[36]: establecer precondiciones y postcondiciones. Con las precondiciones podemos obtener casos de excepción y con las postcondiciones las posibles salidas. Cada postcondición debería al menos definir una prueba.

Veámoslo con un ejemplo: supongamos que lo que debemos probar es una función que obtiene el factorial de un entero *n* pasado como argumento, que no sea mayor a 30. Las precondiciones son que el número no puede ser negativo ni mayor a 30, y la postcondición es que el número se obtiene mediante una productoria de valores enteros en la medida que el argumento sea positivo, o 1 si el argumento es 0.

Bueno: ya con eso tenemos tres pruebas. Una debería chequear que si el número es negativo se reciba una excepción. Otra debería probar con un valor, digamos 5, y verificar que el resultado sea el correcto: 120 en este caso. Y como hay una postcondición excepcional o de borde, correspondiente al argumento 0, este caso también deberíamos probarlo.

Luego, podríamos definir algunas pruebas más para probar cuestiones que a los programadores se le suelen pasar por alto: valores extremos (por ejemplo, números positivos muy altos, y el valor de borde del cambio de condición, que es el argumento 1). Entonces podríamos trabajar con las siguientes pruebas:

Si n = …	**Deberíamos obtener**
-2	Una excepción
0	1
1	1
5	120
30	265252859812191058636308480000000
100	Una excepción

Decimos que cada uno de los valores de *n* que figuran más arriba, salvo el 1, definen clases de equivalencia[37], y por lo tanto debemos tener al menos una prueba para cada una. La prueba de *n = 1* la hicimos porque ejercita un extremo de una clase equivalencia y sospechamos de los

[36] El diseño por contrato es una práctica para la implementación de objetos difundida por Bretrand Meyer y llevada a la práctica en el lenguaje Eiffel [Meyer 1988]. Consiste en considerar a las interacciones de objetos desde el punto de vista de contratos, con precondiciones, postcondiciones e invariantes. Puede verse una explicación más detallada en mis libros de orientación a objetos [Fontela 2010].

[37] La noción de clase de equivalencia se basa en el concepto de relación de equivalencia del Álgebra y la Teoría de Conjuntos. El lector interesado puede profundizar en la literatura especializada.

valores extremos[38].

Diseño de pruebas de cliente

Si la prueba debe chequear un requisito del cliente, y dado que éste debe colaborar en escribirla, es bueno usar la técnica de especificar con ejemplos, para los cuales el rol del programador suele ser el de quien puede prever las clases de equivalencia y valores especiales, y pide ejemplos para cada caso.

En términos de requerimientos de software, pensaríamos que al menos tiene que haber una prueba para el escenario típico, una para cada flujo de excepción y una para cada flujo alternativo.

Cobertura

Llamamos cobertura al grado en que los casos de pruebas de un programa llegan a recorrer dicho programa al ejecutar las pruebas. Se la suele denominar tanto "cobertura de pruebas" como "cobertura de código".

Se usan como una medida, aunque no la única, de la calidad de las pruebas: a mayor cobertura, las pruebas del programa son más exhaustivas, y por lo tanto existen menos situaciones que no están siendo probadas. No obstante, no hay que confundir con la calidad del programa: la cobertura mide sólo la calidad de las pruebas, indirectamente, y sólo subsidiariamente afecta la calidad del programa.

Por lo tanto, el nivel de cobertura no debería guiar el desarrollo. Además, hay que tener cuidado con los análisis de cobertura. Que una determinada línea de un programa esté siendo cubierta por un conjunto de pruebas no implica que se estén analizando todos los casos posibles.

Una buena idea es usar las métricas de cobertura para observar tendencias. Por ejemplo, si una determinada métrica de cobertura disminuye con el tiempo, se puede deber a que la aplicación creció sin que se escribieran las pruebas correspondientes, que algunas pruebas fueron eliminadas, o un poco de cada cosa.

Se ha definido una gran cantidad de métricas de cobertura y en todas ellas se expresa el porcentaje de cobertura: cobertura de sentencias, de ramas o de decisiones, de condiciones, de trayectorias, de invocaciones, etc. Una buena clasificación se puede encontrar en el artículo [Cornett 2014]. La mayor parte de las herramientas de análisis de cobertura mide la cobertura de sentencias, que no siempre es suficiente.

Si bien lograr un 100% parece ser a priori una buena meta, no es así teniendo en cuenta los costos frente a los beneficios que podemos obtener. Habitualmente, un objetivo razonable es llegar a un 80% o 90% en métricas tales como las de sentencias, ramas y combinada de ramas y condiciones. No obstante, hay que tener un especial cuidado con estas recomendaciones. Es común que algunos desarrolladores, con el objetivo de mejorar sus indicadores de cobertura, busquen introducir pruebas sencillas, que casi no agregan valor, pero que mejoren el grado de cobertura reportado.

Finalmente, aun en el marco de un mismo proyecto, suele haber zonas de diferente criticidad.

[38] Esto vale para números y también para otros tipos de argumentos. Por ejemplo, si el argumento fuera una cadena de caracteres, podríamos probar con valores especiales: una cadena en blanco, una vacía, una muy larga, una con caracteres especiales, etc.

Las zonas de mayor criticidad requerirán un grado de cobertura mayor, tal vez cercana al 100%, mientras que las zonas menos críticas tal vez admitan un grado de cobertura mucho menor. Por eso, un análisis serio de cobertura debería partir por hallar partes de un programa que no están siendo recorridas, analizar la criticidad de estas partes y, si lo amerita, agregar pruebas para cubrirlas. En ese sentido, más que métrica de cobertura habría que usar las herramientas para ver cuáles son las líneas que cubrimos y cuáles no.

Hay muchas herramientas de cobertura disponibles, que en general se pueden vincular a los entornos de desarrollo. Entre ellas, destacan Cobertura, Emma, JaCoCo, EclEmma, Clover, SimpleCov, y muchas más.

Recapitulación

En este trabajo hemos analizado las pruebas de software desde el punto de vista de un alumno de programación. Se han presentado las nociones de verificación y validación, y luego se han explicado tipos de prueba, de distinta granularidad, distintos objetivos y diseñadas y ejecutadas por distintos roles.

Como complemento, se han explicado las diferencias entre las visiones tradicionales y otras más modernas, además de exponer brevemente algunas prácticas de desarrollo que pretenden mejorar la calidad del producto antes de llegar a las pruebas.

Bibliografía y referencias

[Adzic 2009] “Bridging the communication gap: specification by example and agile acceptance testing”, Gojko Adzic, Neuri Limited.

[Beck 1999] “Extreme Programming Explained: Embrace Change”, Kent Beck, Addison-Wesley Professional.

[Cornett 2014] “Code Coverage Analysis”, Steve Cornett, consultado en febrero de 2018 en http://www.bullseye.com/coverage.html

[Dijkstra 1972] “The Humble Programmer”, Edsger W. Dijkstra, ACM Turing Lecture 1972.

[Fontela 2010] “Orientación a objetos con Java y UML - 2da. edición”, Carlos Fontela, Nueva Librería.

[Fontela 2012] “Estado del arte y tendencias en Test-Driven Development”, Carlos Fontela, Trabajo de Especialización, UNLP. Disponible en http://sedici.unlp.edu.ar/bitstream/handle/10915/4216/Documento%20completo.pdf?sequence=1 en febrero de 2018.

[Fontela 2013] “Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras”; Carlos Fontela, Tesis de maestría, UNLP. Disponible en http://sedici.unlp.edu.ar/handle/10915/29096 en febrero de 2018.

[Fowler 2011] “SubcutaneousTest”, Martin Fowler, consultado en https://martinfowler.com/bliki/SubcutaneousTest.html en febrero de 2018.

[Gärtner 2012] “ATDD by example: a practical guide to acceptance test-driven development”, Markus Gärtner, Addison-Wesley.

[Hammant 2017] “The Rise of Test Impact Analysis”, Paul Hammant, consultado en febrero de 2018 en https://martinfowler.com/articles/rise-test-impact-analysis.html

[Koskela 2007] “Test Driven: TDD and Acceptance TDD for Java Developers”, Lasse Koskela, Manning Publications.

[McConnell 2004] “Code Complete: A Practical Handbook of Software Construction”, Steve McConnell, Microsoft Press; 2nd edition.

[Meszaros 2007] “xUnit Test Patterns: Refactoring Test Code”, Gerard Meszaros, Pearson.

[Meyer 1988] “Object-oriented software construction”, Bertrand Meyer, Prentice Hall.

[Mugridge 2008] “Managing agile project requirements with storytest-driven development” Rick Mugridge, IEEE software, 25(1).

[North 2006] “Behavior Modification. The evolution of behavior-driven development”, Dan North, Better Software Magazine. Volume-Issue: 2006-03.

[Schwaber 1995] “Scrum Development Process”, Ken Schwaber, OOPSLA'95 Workshop on Business Object Design and Implementation. Austin, USA.

[Suárez 2003] “Documentación y pruebas antes del paradigma de objetos”, Pablo Suárez, Carlos Fontela, FIUBA. Disponible en http://materias.fi.uba.ar/7507/content/20101/lecturas/documentacion_pruebas.pdf en febrero de 2018.