Investigación Científica

PRUEBA FUNCIONAL DEL SOFTWARE

Un proceso de verificación constante

Edgar Serna M.
Autor

iTM

PRUEBA FUNCIONAL DEL SOFTWARE
Un proceso de Verificación constante

Cualquier inquietud o solicitud se recibirá en:

INSTITUTO TECNOLÓGICO METROPOLITANO
Calle 73 76A - 354 – Fondo Editorial
fondoeditorial@itm.edu.co
Medellín, Antioquia.
Colombia

PRUEBA FUNCIONAL DEL SOFTWARE

Un proceso de Verificación constante

FUNCTIONAL TEST OF SOFTWARE

A constant Verification process

Edgar Serna M.

Serna M., Edgar
Prueba funcional de software : Un proceso de Verificación constante / Edgar Serna M. --1a ed.
Medellín : Fondo Editorial ITM, 2013.
246 p. : il. – (Investigación Científica)

Incluye referencias bibliográficas
ISBN 978-958-8743-41-7

1. Ingeniería de software 2. Pruebas de software 3. Desarrollo de software I. Tít. II. Serie

005.1 SCDD 21 ed.

Catalogación en la publicación - Biblioteca ITM

Edición: diciembre 2013
ISBN: 978-958-8743-41-7
Hechos todos los depósitos legales
Publicación electrónica para consulta gratuita

Editado en Medellín, Colombia
Instituto Tecnológico Metropolitano
Calle 73 No. 76A 354
Tel.: (574) 440 5197 ● Fax: 440 5246
www.itm.edu.co

Resumen

El presente libro es resultado de un proceso de investigación en el cual se evaluaron propuestas para el diseño de pruebas funcionales del software. De la muestra inicial se seleccionaron nueve que posteriormente se aplicaron en un proceso experimental para validar, verificar y comparar la eficiencia y eficacia de los casos de prueba que estructuran y generan. Luego se tomaron los resultados para diseñar una nueva propuesta metodológica aprovechando las mejores prácticas. La nueva propuesta se describe y aplica en un estudio de caso mediante el cual se realiza la prueba funcional del software a través de un proceso de verificación paralelo a la Ingeniería de Requisitos.

Abstract

The present book is the result of a research process in which we evaluated the recent proposals to design test cases for functional testing of software. From the initial sample was selected nine that subsequently were applied in a laboratory process to validate, verify and compare the efficiency and effectiveness of test cases that structure. Then were taken the results to design a new methodological approach advantage the best practices of each one. This proposal is described and applied to a study case, by which Functional Testing software is performed through a Verification process parallel verification to the Requirements Engineering.

Prólogo

En la mayoría de las organizaciones dedicadas al desarrollo de software, el programa de pruebas funciona como la barrera final de calidad para una aplicación, permitiendo o impidiendo el paso de un producto de la comodidad del entorno de la Ingeniería de Software al mundo real. Esta función implica una gran responsabilidad porque de ella depende el éxito de una aplicación y, posiblemente, la organización dependa de la calidad del producto software.

El equipo de pruebas debe realizar y gestionar una serie de pequeñas tareas, tantas que, de hecho, muchas se centran exclusivamente en la mecánica de las pruebas de una aplicación de software y prestan poca atención a las tareas circundantes del mismo programa. Cuestiones como la adquisición de datos de prueba adecuados, capacidad de prueba de los requisitos y la arquitectura de la aplicación, estándares y documentación de procedimientos adecuados de prueba, el hardware y las instalaciones, se abordan muy tarde en el ciclo de vida del proyecto. Para proyectos de un tamaño significativo los *scripts* de prueba y las herramientas por sí solos no son suficientes, un hecho que la mayoría de los probadores de software con experiencia puede constatar.

El conocimiento acerca de lo que realmente constituye un esfuerzo exitoso de prueba, de principio a fin, se suele adquirir a través de la experiencia. Comprender que un programa de pruebas puede ser mucho más eficaz si se realizan algunas tareas en las primeras fases del ciclo de vida del proyecto es una valiosa lección.

Las pruebas de software efectivas deben proporcionar prácticas basadas en la experiencia y en los conceptos clave, que la organización puede utilizar para implementar un programa de pruebas exitoso y eficiente. El objetivo debe ser proporcionar una colección destilada de técnicas y discusiones que el equipo de trabajo pueda aplicar directamente para mejorar sus productos y evitar costosos errores y descuidos.

En este libro se describen conceptos específicos para examinar los procesos de Verificación del software, desde la perspectiva del análisis a varias propuestas que aplican la técnica *Casos de prueba desde los casos de uso*, e ilustra la estructura de un importante concepto en las pruebas del software: el esfuerzo de las pruebas se debe integrar linealmente al proceso de desarrollo del producto, desde las fases iniciales del ciclo de vida, para alcanzar logros eficientes y efectivos. Aislar el esfuerzo de la prueba en una caja negra para el final de ese ciclo es un error común y se debe evitar, porque se puede incurrir en un incremento en el tiempo de entrega y de los costos finales.

Es importante anotar que existen factores por fuera del ámbito del programa de pruebas que tienen que ver con el éxito o el fracaso de un proyecto. Aunque el proceso de desarrollo de software sea lo más completo posible, un programa de pruebas bien aplicado ayudará a garantizar el éxito del esfuerzo de la ingeniería. Así mismo, los proyectos también se deben ocupar de cuestiones relacionadas con el negocio, los presupuestos, los tiempos y la cultura de la organización. En algunos casos, estas cuestiones pueden estar en contradicción con las necesidades de un entorno de ingeniería eficaz. Las recomendaciones de este libro suponen que la organización es capaz de adaptarse y de proporcionar el apoyo necesario para el éxito del programa de pruebas; aunque se debe comprender que las pruebas por sí mismas no aseguran la calidad si esta no está presente a través de un proceso de desarrollo efectivo.

El material que contiene el presente trabajo abarca desde procesos y temas relacionados con la gestión, como la especificación de requisitos cambiantes y la composición del equipo de pruebas, hasta aspectos técnicos como las formas de mejorar la capacidad de prueba del sistema y la integración del plan de pruebas al proceso de desarrollo. Pero uno de los aportes más importantes es que se propone a los métodos formales como criterio de aseguramiento de la calidad de las pruebas; el hecho de formalizar la especificación en los casos de uso es innovador, y constituye un adelanto al futuro de las pruebas del software.

Por todo lo anterior, tenemos ante nosotros un texto de calidad superior que se puede utilizar tanto como material para un curso de Verificación y Validación, o como referencia obligada para los equipos de pruebas en las organizaciones. El autor se ha esmerado por ofrecer una combinación precisa entre los conceptos teóricos, la experimentación —ausente en la mayoría de este tipo trabajos—, el análisis certero y la estructuración de una metodología para soportar un nuevo proceso investigativo. Bienvenidos este tipo de trabajos, bienvenidos por su aporte y por su valiosa colaboración a la academia, a la industria y a la comunidad.

Michael Presmann T. (MSTE)
Redmond, Washington

Introducción

La prueba del software es tan antigua en la historia de los computadores digitales como las montañas en la historia de la tierra, y su objetivo es evaluar el software para determinar su calidad. Actualmente, los productos software se encuentran en casi la totalidad de los desarrollos tecnológicos, y la sociedad tiene una alta dependencia de ellos. Debido a esto, y a que el software en sí mismo es un desarrollo tecnológico, la Ingeniería aplicada para desarrollar sus productos debe cumplir cada vez con mayores exigencias de calidad, y entre ellas se encuentran las pruebas que, normalmente, consumen entre el 40 y el 50% del esfuerzo aplicado; además, se incrementan en sistemas que requieren mayores niveles de fiabilidad, por lo que se deben considerar como una parte integral de la Ingeniería de Software.

Con el desarrollo de los lenguajes de cuarta generación, que aceleraron el proceso de implementación de los productos software, también se incrementó la proporción de tiempo dedicado a las pruebas. Debido a que el mantenimiento y la actualización de los sistemas existentes crecen, también se necesita una cantidad más significativa de pruebas para verificar los sistemas luego que se realizan los cambios. A pesar de los avances en los métodos formales y las técnicas de Verificación, los sistemas aún necesitan probarse antes de su uso, por lo que las pruebas se consideran un medio eficaz para asegurar la calidad de un sistema software, así como una de las más complicadas y menos comprendidas áreas de la Ingeniería de Software. Es probable que las pruebas sean aún más importantes en el futuro, aunque actualmente ya son un área importante de investigación en las ciencias computacionales.

La retrospectiva que se hace en el presente libro, acerca de la investigación en Verificación del software, examina el nivel de madurez de las propuestas alrededor de la técnica *Casos de uso desde los casos de prueba*, y hace seguimiento a los resultados de las investigaciones más importantes que han contribuido al crecimiento de esta área. También se examinan los resultados obtenidos mediante aplicación en el laboratorio a un estudio de caso de estas propuestas. Planteada de esta

forma, la prueba del software siempre ha sido un reto, pero el proceso es bien conocido. Alcanzar una combinación de pruebas unitarias, pruebas de integración, pruebas del sistema, pruebas de regresión y pruebas de aceptación, podría contribuir para lograr el objetivo de desarrollar sistemas más fiables. Aunque este esfuerzo será infructuoso si en el proceso de desarrollo del producto no se aplican los estándares de calidad y las mejores prácticas, y si las pruebas no se programan de forma lineal al ciclo de vida.

Este libro es un referente para la mayoría de personas que parecen creer que la prueba del software es una etapa más en el ciclo de vida, y aunque muchos de los planteamientos generales y las técnicas de prueba son los mismos que ofrecen otros trabajos, la adaptación que aquí se hace de esos enfoques y técnicas tradicionales, a los que se suman experiencia y un amplio proceso de investigación, demuestran que algunas cosas pueden hacerse de forma diferente, por lo que presentan entonces nuevos desafíos.

Organización del texto

En este libro se presenta la prueba del software como una actividad de ingeniería que requiere práctica y es esencial para producir software de alta calidad. Está diseñado para utilizarse como texto guía en un curso de introducción a las pruebas en pregrado o como complemento para un curso general sobre Ingeniería de Software, y como un recurso para los ingenieros de pruebas y desarrolladores de software.

Debido a que el texto surge de un proceso de investigación, cuyo objetivo fue "estructurar una metodología genérica para diseñar pruebas funcionales", en el contenido se describen los resultados obtenidos en cada una de sus fases. Entre las posibilidades estudiadas en el desarrollo del proyecto para investigar un área de las pruebas funcionales del software, se seleccionó el análisis. La idea fue estudiar las propuestas recientes para aplicar la técnica *Casos de prueba desde los casos de uso*, debido a su actualidad y porque se consideró con mejor soporte en la literatura, y con la que era posible encontrar resultados con mayor duración e impacto en el tiempo, debido a que se sustenta desde el paradigma de la Programación Orientada por Objetos.

Con este libro se pretendepresentar el paisaje, para muchos complejo y confuso, de los criterios de cobertura de las pruebas con una estructura extremadamente simple. En el campo técnico, las pruebas de software se basan en criterios de cobertura satisfactorios, y la observación central del libro es que existen pocos criterios de cobertura verdaderamente diferentes, ya que cada uno se adapta fácilmente a una de cuatro categorías: gráficos, expresiones lógicas, valores de entrada y estructuras de sintaxis. Esto no solo simplifica las pruebas, sino que también permite un tratamiento teórico conveniente y directo de cada propuesta analizada. Este enfoque contrasta fuertemente con la visión tradicional de las pruebas, en la que se tratan como un proceso diferente en cada fase del ciclo de vida del desarrollo.

Como se afirmó antes, este libro está diseñado y escrito para ser un libro de texto. El estilo de escritura es directo, construye los conceptos desde la base con un mínimo de historia requerida, e incluye ejemplos de aplicación. Proporciona un equilibrio entre la teoría y la práctica, y hace una presentación de las pruebas como un conjunto de actividades objetivas y cuantitativas que se pueden medir y repetir. Utiliza la conceptualización teórica para apoyar las actividades prácticas que los ingenieros de prueba necesitan.

Igualmente, la prueba se concibe como parte de una disciplina mental que ayuda a los profesionales de TI a desarrollar software de mayor calidad, por lo que no es una actividad anti ingeniería, ni un proceso inherentemente destructivo; tampoco es solo para especialistas o «expertos» en el dominio, pero que conocen poco acerca de desarrollo o matemáticas.

El libro está diseñado mediante piezas modulares interconectadas, por lo que se puede utilizar en múltiples cursos relacionados con sus temáticas. La mayor parte del texto solo requiere matemáticas discretas básicas e introducción a la programación, y las partes que necesitan más información están claramente marcadas. Dado que el lector puede ser alguien que esté estudiando para ser ingeniero y cuya meta es producir el mejor software posible al menor costo, los conceptos están fundamentados en la teoría, son prácticos y la mayoría se encuentra actualmente en uso.Para lograr estas características y representarlas, el texto se divide en los siguientes capítulos:

Capítulo 1. *Cuerpo del conocimiento de las pruebas del software.* Aquí se toma el área del conocimiento KA4 del proyecto Swebok y se desarrolla toda su base teórica. El objetivo es presentar una introducción completa al mundo de la prueba del software y conducir al lector a una comprensión sencilla de las temáticas que se aplican más adelante.

Capítulo 2. *Verificación del software.* En el contenido se describe la historia de la Verificación del software como una disciplina amplia y compleja de la Ingeniería de Software, además de la taxonomía respectiva. También se hace un recorrido por la Verificación funcional y se describen las técnicas para llevarla a cabo; al final se presenta un listado de las anomalías que detectan las técnicas descritas.

Capítulo 3. *Estado del Arte.* En este parte del libro se detallan las propuestas que sustentan la técnica Casos de prueba desde los casos de uso, y que fueron encontradas en la amplia revisión a la literatura. Debido a que este texto es resultado de un proceso de investigación, también se describe la aplicación de cada una de ellas a un caso de estudio y se presentan los resultados.

Capítulo 4. *Análisis de objetivos, conceptos y métodos de las propuestas.* Luego de obtener los resultados de la aplicación experimental se seleccionaron los criterios para analizar la eficiencia y la calidad de los casos de prueba obtenidos con cada propuesta. En este capítulo se describen los resultados obtenidos en cuanto a indicadores y métricas y se presentan las conclusiones del análisis. Al final se detallan las buenas prácticas extraídas en el proceso de evaluación.

Capítulo 5. *Segunda fase de análisis.* En este capítulo se presenta un análisis profundo a los resultados del capítulo anterior, se plantea la estructura de las fases como acercamiento a una nueva propuesta para verificar el software con casos de prueba desde los casos de uso, y se describen las fases que un probador debe aplicar para acercarse al problema de un análisis de eficiencia.

Capítulo 6. *Una nueva propuesta metodológica.* Luego de aplicar, comparar y analizar las propuestas seleccionadas, con los resultados se estructura una nueva propuesta para generar casos de prueba desde los casos de uso, orientada a verificar los aspectos funcionales del software. En este capítulo se describen las fases propuestas y se ejemplifican en un estudio de caso.

Anexo I. *Aplicación de la propuesta a un estudio de caso.* En este anexo se aplica la nueva propuesta a un estudio de caso con lo que se complementa el capítulo 6.

Anexo II. *Herramientas para diseñar casos de prueba.* En este anexo se presenta un listado de las herramientas que automatizan el diseño de casos de prueba, un área que no se cubre en este texto pero que se plantea como trabajo futuro.

Reconocimientos

La mayoría de las contribuciones para la edición de este libro fueron hechas y revisadas de forma anónima. El autor expresa especial agradecimiento a los revisores por sus valiosas observaciones y por su rápida respuesta, lo que permitió alcanzar la culminación oportuna, también a todos los autores que publicaron los excelentes trabajos que se analizan y aplican en este texto; al maestro Fernando Arango Isaza, profesor de la Universidad Nacional de Colombia sede Medellín, por su indirecta y valiosa ayuda en la preparación del texto. Al Instituto Antioqueño de Investigación (IAI) por el patrocinio que brindó a la realización de la investigación. Por último, reconocer el aporte delos ingenieros Jairo Humberto Betancur y Gustavo Londoño, quienes participaron como estudiantes practicantes en la primera fase del proyecto.

Contenido

Capítulo 1

Cuerpo del conocimiento de las pruebas del Software

A pocos desarrolladores les gusta hacer pruebas; incluso a muy pocos especialistas en diseño les llama la atención hacerlas debido a que consumen mucho tiempo del proyecto y por lo menos la mitad de la mano de obra utilizada para desarrollar un producto software (Brown y Lipow, 1973; Wolverton, 1974; Boehm, 1975; Goodenough, 1979; Radatz, 1981).Esta actitud es comprensible: el software es efímero, no es un objeto del mundo físico, yen el fondo la mayoría de personas no creen en él -al menos no de la forma que lo hacen con el hardware-. Si es insustancial, entonces ¿cuánto más insustancial es la percepción de las pruebas? Ni siquiera se encuentran indicios de querer depurar el código para realizar los ensayos de prueba, porque el esfuerzo realizado en ellos parece perdido cuando las pruebas no revelan errores.

Existe otro problema con las pruebas-más profundo- relacionado con la razón por la cual se aplican: se hacen para encontrar errores (Miller, 1979; Myers, 2004). Se ha generado un mito según el cual la programación es buena cuando no genera errores, pero si solo pudiéramos realmente concentrarnos, si todo el mundo utilizara el mismo modelo de programación, el diseño *top-down* olas tablas de decisión; si todos los programas fueran escritos en *Squish,* si tuviéramos derecho a las balas de plata, entonces no habría errores. Así va el mito: *existen errores porque hacemos algo mal, y si lo que hacemos está mal, debemos sentirnos culpables al respecto.* Por lo tanto, la prueba y el diseño de la prueba ascienden a una admisión de fracaso, que infunde una buena dosis de culpabilidad en la que el tedio de las pruebas es solo el castigo por nuestros errores. Castigo, ¿por qué? ¿Por ser humanos? Culpa, ¿de qué? ¿Por no lograr la perfección inhumana? ¿Por no distinguir entre lo que piensa un usuario y lo que expresa? ¿Por no ser telepáticos? ¿Por no resolver los problemas de comunicación humana que han planteado los filósofos desde hace más de 40 siglos?

Las estadísticas demuestran que un desarrollo bien hech otendrá entre uno y tres errores por cada cien declaraciones (Akiyama, 1971; Shooman y Bolsky, 1975; Boehm, 1975;Endres, 1975; Alberts, 1976; Thayer, 1976; Radatz, 1981; Weiss y Basili, 1985). Ciertamente, si tenemos un 10% de tasa de error entonces lo que necesitamos es más formación en Ingeniería de software, o admitamos que nos merecemos una amonestación yaceptemos la culpa. Existen personas que afirman poder escribir programas libres de errores, pero con el desarrollo y los errores sucede lo siguiente: yo los tengo, tú los tienes, todos los tenemos; el punto es evitarlos y descubrirlos lo antes posible, pero no sentirnos culpables por ellos.

Esta es una situación desconcertante que ha hecho que en las universidades y empresas de Europa y Asia trabajen en busca de soluciones, y que adquieran más experiencia en el valor de las pruebas que en países como EE.UU., donde solo tres universidades de cada veintelas contemplan en sus planes formativos, mientras que en Europa la relación es siete de cada diez. La razón de esta diferencia se debe a una cultura académica y de negocios,en la que el valor de la prueba no sevende como una solución ampliamás que en los libros básicos acerca del tema.

Las pruebas de software consisten en verificar dinámicamente la conducta del programa bajo un conjunto finito de casos de prueba, y comparar los resultados con lo que se esperaba(Weiss y Basili, 1985).Esta área del conocimiento se divide enotras dos:(1) la que se organiza conforme a las fases tradicionales para testeo de grandes sistemas software y (2) la que trata las pruebas para condiciones o propiedades específicas.

1. El área de las pruebas del software del Proyecto Swebok

A pesarde que en todo el mundoexisten millones deprofesionales desoftware,y delapresencia generalizada de sus desarrollos en nuestra sociedad, solo recientemente la Ingeniería de Software ha alcanzado el estatus de una legítima disciplina de ingeniería y de una profesión reconocida. Lograr consenso para esta profesión en un cuerpo básico de conocimientos es un hito clave en todas las disciplinas y ha sido identificado por la IEEE Computer Society como crucial para elevar esta Ingeniería a un nivel profesional. La Guide to the Software Engineering Body of Knowledge SWEBOK (Abran *et al.*, 2004) escrita bajo los auspicios del Professional Practices Committee, hace parte de un proyecto de varios años diseñado para lograr ese consenso.

La IEEE Computer Society define a la Ingeniería de Software como la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software; es decir, la aplicación de Ingeniería al Software (IEEE, 1990). Para que esta área del conocimientoseaconocida plenamente como una verdadera disciplina de ingeniería y como una profesión, es imprescindible el consenso sobre un cuerpo básico de conocimientos. Starr (1982, p. 98) ilustra bien este hecho cuando define las pautas para reconocer cuándo una disciplina es legítima y además una profesión:

> La legitimación de la autoridad profesional consta de tres demandas distintas: (1) que el conocimiento y las competencias de los profesionales hayan sido validados por una comunidad,(2) que ese conocimiento, validado por consenso, se base en cimientos racionales y científicos y (3) que el juicio y asesoramiento de los profesionales esté orientado hacia un conjunto de valores sustantivos, tales como la seguridad. Estos aspectos de la legitimidad corresponden a los tipos de atributos: colegial, cognitivo y moral, usualmente consagrados en el término «profesión».

Ford y Gibbs (1996) estudiaron varias profesiones reconocidas, incluidas la medicina y la ingeniería, y llegaron a la conclusión de que una profesión enesta última se caracteriza por:

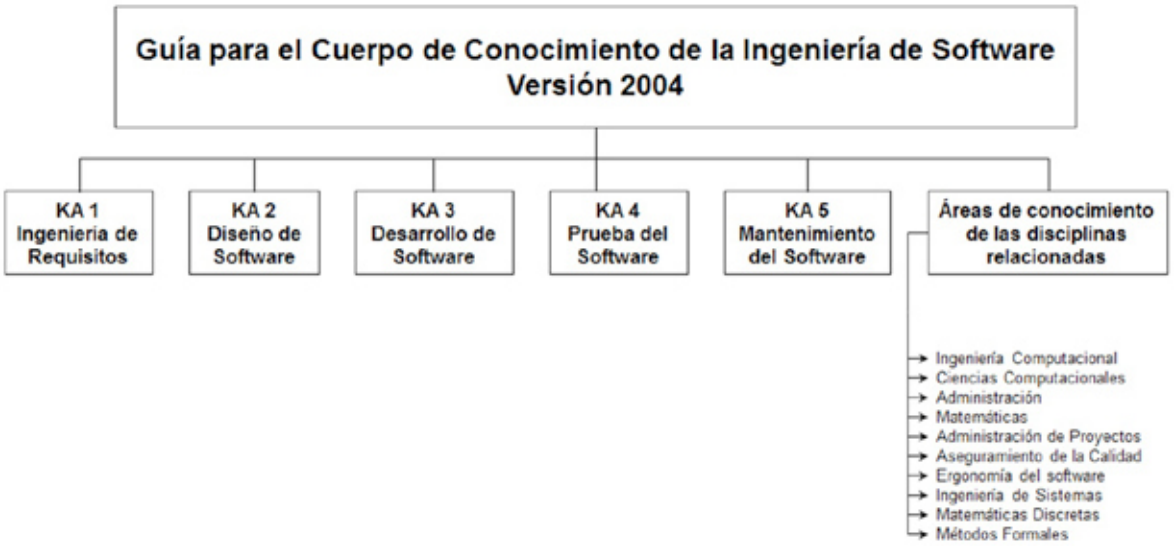
- Contar con una formación profesional, con base en un plan de estudios validado por la sociedad a través de una acreditación
- Matricular una aptitud para ejercerla, por medio de una certificación voluntaria o de un licenciamiento obligatorio
- Desarrollar habilidades especializadas ymantener una formación profesional continua
  Ser reconocida públicamente, a través de una sociedad profesional
- Tener un compromiso de respeto por las normas de conducta prescritas en un código de ética

Articular un cuerpo de conocimientos de la Ingeniería de Software y lograr el mayor consenso posible sobre su contenido, es un paso esencial hacia el desarrollo de una profesión, y SWEBOK representa el amplio acuerdo acerca de lo que debe saber un ingeniero de software profesional. Sin ese consenso, no se puede validar el conocimiento, los currículosno podrían preparar a un individuo para una certificación, y no se pueden formular criterios para acreditar un programa de estudios. El desarrollo del consenso es también un requisito previo a la adopción del desarrollo de capacidades coherentes y programas de formación profesional en las organizaciones. IEEE Computer Society ha defendido el desarrollo de tal consenso internacional sobre un compendio y una guía para el cuerpo de conocimientos que ha evolucionado y se ha desarrollado en las últimas cuatro décadas: el proyecto Guide to the Software Engineering Body of Knowledge (SWEBOK).

SWEBOK es un proyecto cuyo objetivo es describir la suma de conocimientos dentro de la Ingeniería de Software como profesión. Debido a que usualmente no es posible describir todo el cuerpo deconocimiento en un único documento, incluso de una disciplina emergente como la Ingenieríade Software, existelanecesidad deunaGuía para el cuerpo de conocimiento de la Ingeniería de Software. El objetivo de esta guía es identificar y describir qué subconjunto del cuerpo de conocimientos es aceptado generalmente,aunque los ingenierosde softwaredeben estarbien informados, no solo en Ingeniería de Software, sino también, por supuesto, en otras disciplinas relacionadas. Además, SWEBOK proporciona un marco estructurado por cada área de conocimiento dividido en sub-áreas, temas y subtemas. El objetivo de esta estructura es permitirles a los lectores encontrar rápidamente sus temas de interés y proporcionarles una referencia para

ese conocimiento. También incluye lecturas complementarias y normas aplicables. El ámbito de aplicación de los conocimientos cubiertos es el que se considera que debe ser generalmente aceptado, reconociendo que se excluye lo especializado, lo innovador y los conocimientos de tipo investigativo.

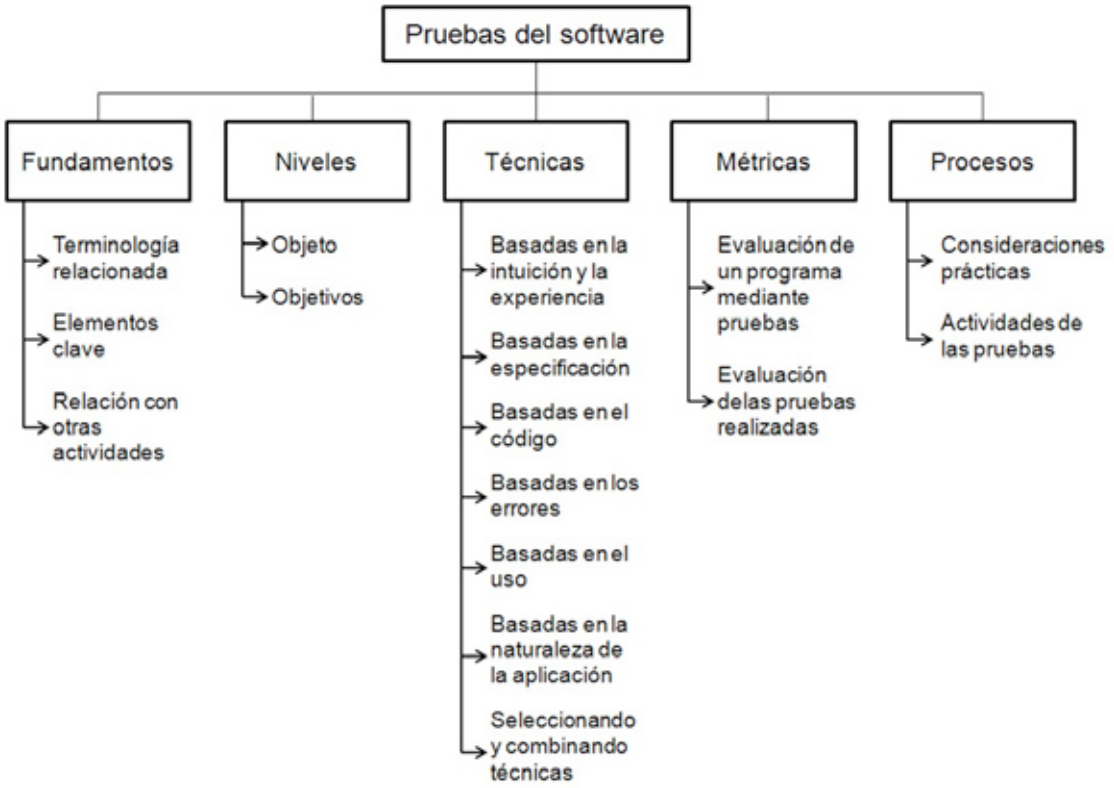
Figura 1. Áreas de conocimiento de SWEBOK (Abran et al., 2004)



SWEBOK se subdivide en diez áreas de conocimiento -KA- como se observa en la Figura 1, y para proporcionar un acceso a los conocimientos de actualidad, cada área se subdivide en temas y subtemas; de esta forma identifica el material de referencia principal relacionado ypara los temas listados. *En el Paradigma Orientado por Objetos*, las 10 áreas de conocimiento se podrían considerar como sub-clases de la super-clase SWEBOK. Todos los conceptos acerca de la Ingeniería de Software serían una sub-clase de una o más áreas de conocimiento. Esto significa que un concepto debe ser una sub-clase de la súper-clase y tener relaciones a diferentes áreas de conocimiento. Pero las súper-clases y las sub-clases, así como las definiciones de los conceptos, representan solo un primer paso. Un usuario está interesado no soloen las definiciones de los conceptos, sino también en la información más detallada acerca de los temas que son importantes para él. En SWEBOK este nivel de detalle de la información no está en la guía sino en la lista de referencias aprobadas internacionalmente.

En este textose desarrolla el área de conocimiento KA 4: Pruebas de Software, que se esquematiza en la Figura 2. A continuación se describe cada uno de estos conceptos, y las técnicas para probar el software se explican con mayor detalle en el numeral 4 de este capítulo.

Figura 2. Sub-áreas y temas del área Pruebas del Software (Abran et al., 2004)



1.1 Pruebas del software

El software no es diferente de otros procesos físicos en los que se reciben los insumos y se elaboran productos, es diferente en la forma en como falla. La mayoría de los sistemas físicos fallan en un fijo y razonablemente pequeño conjunto de formas, mientras que el software puede fallar de muchas y extrañas formas, por lo que la detección de todas es en general inviable. A diferencia de los sistemas físicos, la mayoría de los defectos en el software son errores de diseño, no defectos de fabricación. El software no sufre corrosión, desgaste o vencimiento, y por lo general no cambia hasta que se actualiza o se vuelve obsoleto. Así que, una vez entregado, los errores de diseño permanecerán latentes hasta su activación.

La prueba del software es un proceso en el que se ejecuta un programa o sistema con la intención de encontrarle errores (Myers, 2004). Es cualquier actividad dirigida a evaluar un atributo o capacidad de un programa o sistema, y la posibilidad de determinar que cumple su especificación(Hetzel, 1988), es decir, las necesidades del cliente. Los errores casi siempre existen de forma moderada en cualquier módulo, y no porque los desarrolladores sean descuidados o irresponsables, sino porque la complejidad del software es en general intratable -los seres humanos tienen una capacidad limitada para gestionar la complejidad-. También es cierto que para cualquier sistema complejo los errores de diseño nose debendescartar completamente. De la misma forma,

descubrir errores en el diseño de software es difícil por la misma razón de su complejidad. Dado que el software y los sistemas digitales no son continuos, los valores límite de las pruebas no son suficientes para garantizar su corrección: todos los posibles valores de entrada tienen que ser probados y comprobados, pero la prueba completa es imposible. Por ejemplo, la prueba exhaustiva de un programa simple, con una entrada de dos datos enteros de 32 bits -264 distintos casos de prueba-, tomaría demasiado tiempo incluso si la prueba se ejecutara a un ritmo de miles casos por segundo. Obviamente, para un módulo de software real la complejidad puede ir mucho más allá de este ejemplo. Si se consideran los aportes del mundo real, el problema se agravará porque el calendario y los efectos imprevisibles del medio ambiente y las interacciones humanas hacen parte de los posibles parámetros de entrada que se deben probar.

1.2 Fundamentos de la prueba del software

El estudio de las pruebas de software comienza con la descripción y comprensión del vocabulario esencial relacionado con dicho tema en el que se cubren las definiciones básicas del área, la terminología, los términos clave ysus relaciones con otras actividades. El conocimiento de estas condiciones básicas es necesario para asegurar que se comprenda el tema de las pruebas de software, además, para que se conozca el vocabulario común ampliamente aceptado por la comunidad.

A continuación se presenta un conjunto de definiciones básicas de términos que se utilizan en este texto, comunes en los documentos que abordan el tema de laspruebas de software. Otras definiciones adicionales aparecen en el contenido como conceptos de ayuda a la comprensión del área, y otras se basan en los términos descritos por la IEEE en la Colección de Normas de Ingeniería de Software (IEEE, 1993). Esta colección incluye las normas estándar IEEE en su Glosario de Terminología (IEEE, 1990), un diccionario dedicado a la descripción del vocabulario de la Ingeniería de Software que contiene definiciones de términos ampliamente utilizados.

1.2.1 Terminología relacionada con la prueba del software

- **Error.** Es un mal funcionamiento que aparece durante la ejecución de un producto software. Puede aparecer en cualquier momento de su uso e impedirun correcto funcionamiento del producto, o en cualquiera de las fases del ciclo de vida, aunque los más sobresalientes se detectan en la Ingeniería de Requisitos y en la fase de Desarrollo. Se produce como manifestación a una anomalía del software:un descuido, una idea errónea o una incomprensión de los requisitos por parte del desarrollador. Por ejemplo, puede malinterpretar una notación de diseño o podría escribir un nombre de variable de forma incorrecta. En la categoría de desarrollador de software se incluye a ingenieros, desarrolladores, analistas y probadores.

- **Falla o defecto.** Una falla se introduce en el software como resultado de un error. Es una anomalía que puede provocar un comportamiento no planificado en la especificación. Las fallas o defectos a veces se denominan "bugs", pero el uso de este término trivializa su impacto sobre la calidad del software. El uso del término «defecto» se asocia también con otros componentes del software, como los requisitos y los documentos de diseño; los fallos que se producen en estos componentes también son causados por errores,que por lo general se detectan mediante un proceso de revisión.

- **Fracasos.** Es la incapacidad de un sistema o componente de software para realizar las funciones especificadas en los requisitos (IEEE,1990). Un comportamiento incorrecto puede incluir valores erróneos para las variables de salida, una respuesta incorrecta de un dispositivo o una imagen incorrecta en un cuadro de diálogo. Los probadores detectan los errores, y los desarrolladores localizan y reparan las fallas. Cuando el software está en funcionamiento los usuarios, o el equipo de soporte, pueden encontrar fracasos que reportan a losdesarrolladorespara su reparación.Una falla en el código no siempre produce un fracaso. De hecho, las fallas del software pueden estar latentes durante un largo período de tiempo sin exhibir ningún comportamiento incorrecto, sin embargo, cuando se producen las condiciones adecuadas la falla se manifestará como un fracaso. Voas (1990) describe esas condiciones:

  1. Unaentrada al software debe causar la ejecución de una declaración defectuosa.
  2. Una declaración defectuosa debe producir un resultado diferente al de una declaración correcta. Este evento produce un estado interno incorrecto para el software.
  3. El estado interno incorrecto debe propagarse a la salida, de modo que el resultado del fracaso se pueda observar.

  Cuando el software revela fácilmente sus fallas como fracasos es elementalmente comprobable; este es un atributo deseado porlos probadores quienes necesitan trabajar con los diseñadores para asegurar la comprobación del software.

- **Casos de prueba.** El enfoque habitual para la detección de erroreses que el probador seleccione un conjunto de datos de entrada, y a continuación ejecute el software con esos datos bajo unas condiciones particulares. Para poder decidir si el software pasa o no la prueba, necesita conocer,dados el conjunto de entradas y las condiciones de ejecución, cuáles son las salidas correctas del programa. Luego empaqueta esta información en un ítem llamado caso de prueba. En la práctica, un caso de prueba es aquel en el que una prueba relacionada con un tema contiene la siguiente información:

  1. Un conjunto de datos de entrada, recibidos por el código bajo prueba desde una fuente externa que puede ser hardware, software o personas.

2. Condiciones de ejecución requeridas para efectuar la prueba, por ejemplo, cierto estado de la base de datos o la configuración de un dispositivo hardware.
3. Salidas esperadas, que son los resultados específicos esperados al ejecutar el código bajo prueba.

Esta es la información mínima que debe contener en un caso de prueba, y se basa en la descripción de IEEE (1990). Las organizaciones pueden decidir la información adicional que se debe incluir en los casos de prueba para incrementar su valor como objeto reutilizable, o para proveer información más detallada para probadores y desarrolladores. Por ejemplo,la prueba de un componente objetivo se podría incluir para expresar objetivos de prueba, como ejecutar un grupo particular de líneas de código o comprobar que se ha logrado un determinado requisito. En el diseño de una prueba de especificación, que describa con precisión el contenido de cada caso de prueba, deben participar los desarrolladores, los probadores y el grupo de aseguramiento de la calidad del software. El contenido y los formatos deben aparecer en los estándares de documentación de pruebas de la organización.

- **Prueba.** Es un conjunto de casos de prueba y los procesos relacionados necesarios para ejecutar un plan de pruebas. Algunas veces, un grupo de pruebas relacionadas se refiere a un conjunto de pruebas que están asociadas con una base de datosy que usualmente se ejecutan al mismo tiempo;tambiénse conoce como suite de pruebas (Beizer, 1990).

- **Oráculo de prueba.** Un oráculo es cualquier agente-humano o mecánico-,un documento o un programa que puede decidir si un producto software responde correctamente a una prueba, y si lasupera o no.También es un programa o documento que produce o especifica el resultado esperado de una prueba (Howden, 1981). Su estructura incluye una especificación -especialmente una que contenga pre y post-condiciones-, un documento de diseño, y un conjunto de requisitos.

- **Banco de pruebas.** Es un ambiente que contiene el hardware y el software necesarios para probar un componente o sistema. Incluye todo el entorno de la prueba, por ejemplo: simuladores, emuladores, verificadores de memoria, probadores de hardware, herramientas de software, y todos los ítems necesarios para soportar la ejecución de las pruebas.

- **Calidad del software.** De acuerdo con IEEE (1990) calidad es:

1. El grado en que un sistema, componente o proceso cumple con los requisitos especificados.
2. El grado en que un sistema, componente o proceso cumple con las necesidades o expectativas de un cliente o usuario.

Para determinar si un sistema, subsistema o proceso es de calidad se utilizan los atributos de calidad-características que reflejan la calidad-. Mediante las métricas de calidad se puede medir el grado que posee un determinado atributo del sistema. Estas métricas son una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado(IEEE, 1990). Existen métricas de productos y procesos, como el tamaño del software, usualmente medido en líneas de código, o el tiempo requerido para ejecutar una tarea. Las métricas también son unamedida cuantitativa del grado en que un elemento posee un determinado atributo de calidad (IEEE, 1990). Los diferentes atributos de calidad se describen en Grady(1992), Schulmeyery McManus(1999), ISO/IEC(2001) y Suryn y Abran (2003):

- Correctitud. Grado en el que un sistema realiza la función pretendida
- Confiabilidad. Grado en el que se espera que el software desempeñe las funciones requeridas, sobre condiciones determinadas,en un período de tiempo determinado
- Usabilidad. Grado de esfuerzo necesario para aprender, operar, preparar e interpretar salidas del software
- Integridad. Se refiere a la habilidad del sistema de resistir tanto ataques intencionales como accidentales
- Portabilidad. Es la capacidad del software de ser transferido de un ambiente a otro
- Mantenibilidad. Es el esfuerzo necesario para hacer cambios en el software
- Interoperabilidad. Esfuerzo necesario para relacionar o emparentar un sistema con otro
- Testibilidad. Es un atributo de calidadque interesa más a los desarrolladores y probadores que a los clientes, y se puede expresar de dos formas:

1. La cantidad de esfuerzo necesario para probar el software y garantizar que dicha prueba se realiza de acuerdo con los requisitos especificados -en relación con el número de casos de prueba necesarios-

2. La capacidad del software para revelar errores bajo condiciones de prueba algunos programas están diseñados de tal manera que las anomalías se encuentran ocultas en condiciones normales de prueba-

Para asegurar que se aborden las cuestiones de testibilidad, los probadores deben trabajar con los analistas, los diseñadores y los desarrolladores durante todo el ciclo de vida del proyecto.

- **Reseñas.** Contrario a las técnicas dinámicas de prueba basadas en la ejecución, usadas para detectar errores y evaluar la calidad de software, las reseñas son un tipo de técnica estática de prueba que se puede utilizar para evaluar la calidad de un programa, un documento de requisitos, un plan de pruebas, un documento de diseño, un componente de código u otro tipo de documentos.

1.2.2 Elementos clave de las pruebas del software

En el ciclo de vida del software, las pruebas intentan exponer la mayor cantidad de errores que sea posible, con el objetivo de brindarles al equipo y al usuario la tranquilidad de que el productoesfiable. La meta ideal es garantizar completamente su correctitud, pero en todos los casos, salvo algunos muy simples, es imposible llevar a cabo todo el conjunto de pruebas para un conjunto finito de datos (Howden, 1976; Weyuker y Ostrand, 1980).

Goodenough y Gerhart (1977) definen un conjunto ideal decasos de prueba que tiene la característicade exponer todos los errores en un programa. Así, si un programa produce resultados correctos para un conjunto de casos de prueba, entonces debe estar correcto. Sin embargo, estas propiedades no son constructivas en el sentido de que no dicen cómo producir pruebas ideales para un determinado programa. Además, generalmenteno es posible determinar si un conjunto de pruebas para un programa es ideal, y tampoco se puede garantizar que un programa sea correcto aun cuando no se le hayan detectado errores.

Se han tratado de especificar algunas cuestiones clave como base necesaria del trabajo en pruebas de software, y el proyecto SWEBOK propone algunas de ellas de acuerdo con los siguientes criterios:

- **Criterios de selección/idoneidad.** Se usan para determinar cuáles son los casos de prueba adecuados para ejecutar, o para comprobar si la implementaciónse puede utilizar para llevar a cabo las pruebas. Se pueden usar también para seleccionar casos de prueba o para comprobar si es apropiado el grupo seleccionado, es decir, para decidir cuándo terminar las pruebas(Weyuker et al., 1991; Zhu et al., 1997; Pfleeger, 2001).

- **Efectividad/Objetivos.** Para determinar la efectividad de un conjunto de pruebas es necesario tener claro sus objetivos: no se trata de observar la ejecución de un programa por el simple hecho de revisarlo, sino de hacerlo con unos objetivos trazados y unas metas por cumplir, porque realizar pruebas es ejecutar una serie de componentes del programa. Por esto existen diferentes objetivos que pueden ayudara seleccionar el conjunto de pruebas (Beizer, 1990; Perry, 1995; Frankl et al., 1998).

- **Realizar pruebas para identificaranomalías.** Este enfoque de identificación consiste en que las pruebas son exitosas si, como resultado, se encuentran errores en el sistema. Este enfoque es contrario al de realizar pruebas con el objetivo de demostrar que el software satisface los requisitos,en el que el éxito de la prueba consiste en que, como resultado, no arroja errores o por lo menos no son significativos (Beizer, 1990; Kaner et al., 1999).

- **El problema del oráculo.** El problema del oráculo consiste en poder determinar si un procedimiento de decisión se puede definir para la interpretación de los resultados de las

pruebas, de acuerdo con una especificación formal. En este contexto, la prueba consiste en comprobar si los programas satisfacen los axiomas de especificación. En consecuencia, las operaciones de ejercicios de pruebas mencionados por los axiomas y los oráculos evalúan los primeros de acuerdo con los resultados arrojados por las pruebas. Automatizar este elemento es costoso, por lo que su utilización se limita a programas extensos o sistemas de alta complejidad y seguridad (Weyuker, 1983; Beizer, 1990).

- **Limitaciones teóricas y prácticas.** En muchas ocasiones los equipos de prueba se exceden en confianza al determinar que se supera una serie de casos de prueba, algo que la teoría relacionada ha advertido. Pero, la teoría de las pruebas establece principalmente resultados negativos, porque establece aquello que la prueba no puede conseguir, en vez de lo que ha conseguido. En este sentido se pronunciaba Dijkstra(2000, p. 4) al afirmar que las pruebas de un programa se pueden usar para demostrar la presencia de errores, pero nunca para demostrar su ausencia. Esto resulta obvio al aceptar que aplicar un conjunto completo de casos de prueba no es posible, mucho menos en el software real, lo que origina que las pruebas se deban dirigir en función de los riesgos, y correr el «riesgo» de que se vean como una estrategia de gestión de riesgos(Howden, 1976; Kaner et al., 1999).

- **El problema de los caminos no alcanzables.** Son aquellas secuencias de caminos de ejecución que no se pueden ejecutar para ninguna entrada de datos. Es un problema importante en las pruebas orientadas por caminos, y particularmente en las derivaciones automáticas de entradas de casos de prueba que se emplean en las técnicas basadas en el código(Abranet al., 2004). Por esta razón, las pruebas se deben programar e incluir en todo el ciclo de vida del producto, ya que no es posible alcanzar el cubrimiento total, y tampoco es recomendable ejecutar pruebas improvisadas.

- **Posibilidad de hacer pruebas.** Este término tiene dos significados que se relacionan pero que en esencia son diferentes: (1) se refiere al grado de facilidad del software para satisfacer un determinado criterio de cobertura de pruebas (Goodenough y Gerhart, 1977) y (2) es la probabilidad, posiblemente cualificada estadísticamente, de que las anomalías del software queden expuestas durante las pruebas, cuandoson errores (Bache y Müllerburg, 1990; Voas y Miller, 1995; Bertolino y Marrè, 1996).

1.2.3 Las pruebas y su relación con otras actividades

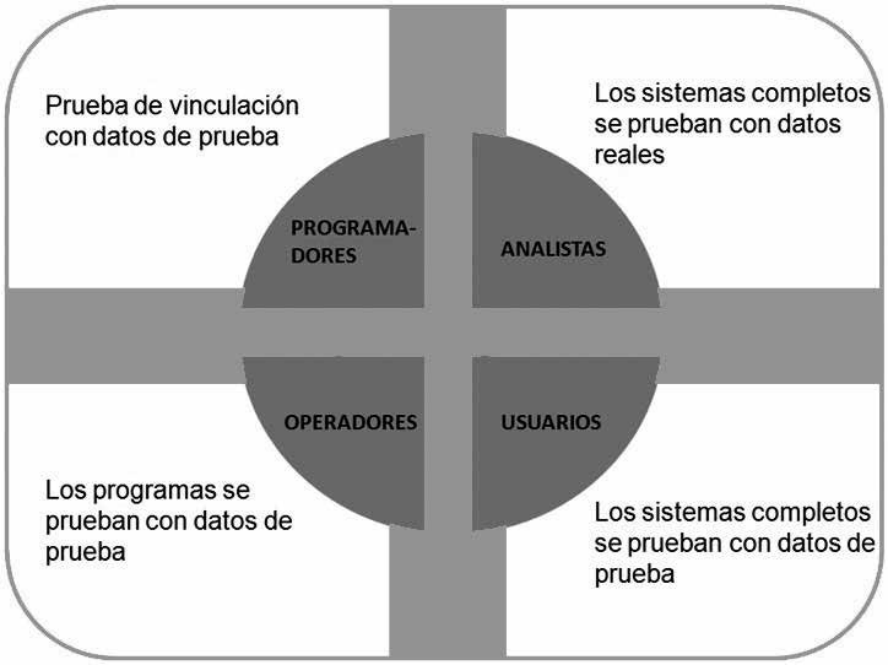
Para que los programas de aplicación del sistema, escritos o modificados, se prueben lo más ampliamente posible, la idea es integrar las pruebas a cada una de las fases del ciclo de vida en el proceso de la Ingeniería de Software-se debe evitar probar al azar y por tanteo-. Además, como se ha mencionado antes, las pruebas se planifican, estructuran y ejecutan en todo el proceso de

desarrollo del sistema, no al final de él. El objetivo esdescubrir errores desconocidos en cada momento del proceso, no demostrar la perfección de los programas.

Para muchos profesionales en ciencias computacionales es tedioso ejecutar pruebas,aunque entienden que es una serie de pasos que ayudan a asegurar la calidaddel producto.De alguna manera se puede tener cierto grado de tranquilidad cuando en el proceso se realizan pruebas al sistema, porque no es práctico esperar a que falle para corregirlo luego de la implementación. A medida que avanza el desarrollo se debe aplicar la pruebaa lossubsistemas o módulos del programa, en niveles diferentes y en varios intervalos; de esta manera se aseguramedianamente que el cliente recibirá un programa más cercano a lo que esperaba; por lo que es necesario, antes que el sistema se ponga en producción, quetodos los programas se verifiquen en el escritorio.

Como se observaen la Figura3, desarrolladores, analistas, operadores y usuarios tienen una función diferente en lo que respecta a los aspectos que se deben probar. Las pruebas de hardware normalmente se proporcionan como servicio de parte de los proveedores, quienes ejecutan sus propias pruebas antes y durante la entrega.

Figura 3. Las pruebas y su relación con otras actividades (Sommerville, 2006)



La mayor parte de las pruebas es responsabilidad del encargado de la fase en ejecución, pero en cada una de ellas se debe aportar para que el plan de pruebas se lleve a cabo secuencialmente. En el análisis, el analista sirve comoconsejero y coordinador; su función es asegurar que los desarrolladores implementen las técnicas de prueba adecuadas,pero probablemente no desempeñe personalmente este nivel de verificación. Los desarrolladores deben hacer pruebas de escritorio para verificar la forma en que los módulos del sistema funcionarán al integrarse. El analista verifica

la salida en busca de errores a lo largo de este proceso,e informa al desarrollador de cualquier corrección que sea necesaria. El analista normalmenteno recomienda o crea datos de prueba, pero señalalas omisiones de tipos de datos que se deberán agregar en las pruebas posteriores.

- **Gestión de la calidad del software.** La gestión de la calidad del software -SQM por sus siglas en inglés-es un producto que resulta luego de aplicar todas las perspectivas de procesos, productos y recursos del software, y está conformado por una serie de actividades orientadas a las pruebas, algunas de las cuales pueden encontrar errores directamente mientras que otras indican dónde implementar más revisiones. Planificar la calidad del software implica:(1) definir el producto requerido en términos de sus características de calidad y(2)planificar los procesos para alcanzar el producto requerido.

  Los procesos de SQM deben verificar: (1) cómo satisface el producto software al cliente y a los requisitos del personal implicado,(2) cómo proporcionar valor a los clientes y demás personal implicado y(3) cómo proveer la calidad precisa de software para conocer los requisitos deeste (Arthur, 1997).Algunos de los procesos específicos SQM están definidos en ISO/IEC (1995), que además corresponden a las técnicas de prueba y al Aseguramiento de la Calidad:

  - Procesos de aseguramiento de calidad
  - Procesos de Verificación
  - Procesos de Validación
  - Procesos de revisión
  - Procesos de auditoría

- **Pruebas de validez y verificación formal.** La solución a un problema real depende en gran medida delos niveles de comprensión, abstracciónу modelado, de que la solución algorítmica corresponda a una solución real y de que todo el procesose haya implementado correctamente. Aunque es evidente que existen muchos puntos en donde se puede fallar. La teoría y la práctica del desarrollo de software sugieren dos caminospara realizar este proceso, no necesariamenteexcluyentes: (1) Verificar, que consiste en demostrar que la solución satisface los requisitos y (2) Validar, que escomprobar si la solución se comporta de la manera esperada para todos los casos. Tanto el uno como el otro necesitan un referente para contrastar la solución contra unos requisitosque se deben cumplir, por lo que es necesario que estos se especifiquen claramente. Para lograrlo, se recurre a herramientas de lógica matemática especialmente diseñadas para las ciencias computacionales, igualmente a los métodos formales de especificación, al modelado, a la verificación y a la prueba, que son necesarios para poder afirmar con cierta certeza si la solución es funcional (Perry, 2000).

  Depuración. Es un concepto que está estrechamente relacionado con la calidad, ycuya finalidad es depurar un programa para librarlo de los errores más o menos graves; además, en programas

grandes y complejos es un proceso costoso y arduo, pero que es importantepara cumplir con el objetivo de entregar un producto con cierto nivel de calidad.

De forma muy general, la depuración del software se realiza mediante un proceso en el que se busca la «perfección» del producto, es decir que esté libre de errores e inconvenientes graves. Esto se logra mediante una serie de transformaciones sucesivas: (1) aplicar al programa casi terminado las pruebas iniciales, unitarias o de laboratoriopara comprobar que su funcionamiento corresponde con el esperado, y que está «libre» de errores-esta labor la realiza el desarrollador antes de entregar el producto-, (2) seleccionar otros probadores -beta testers- para que ejecuten otra serie de pruebas y para verificar que el producto se comporta de forma más o menos aceptable -esta tarea se ejecuta cuando el desarrolladordetermina que el software está suficientemente probado y nole encuentra más errores aparentes- y (3) con la información, opiniones y sugerencias de los beta testers, se hace el lanzamiento de la primera versión pública del producto –*release*-. Este proceso continúa con la recolección de sugerencias, cuestiones y posibles errores que hayan pasado inadvertidos hasta el momento, oque reportan los usuarios,y que se deben implementar o corregir en las versiones sucesivas.

- **Programación.** Desarrollar implica ejecutar dos tipos de pruebas: las pruebas unitarias y las pruebas de integración. Su propósito es reducir la brecha existente entre el tiempo en el que se introducen errores en el código y el tiempo en el que se detectan. En este tipo de pruebas se incluye un subconjunto de pruebas específicas que no involucraa las pruebas del sistema, alfa,beta, de estrés, de construcción, de posibilidad de uso, u otras más especializadas.

- **Certificación.** La perspectiva de software certificado es dudosa para muchos. Los proveedores de software garantizan que los discos están libres de errores, pero no pueden hacer lo mismo con el software contenidoen ellos, ni que es el adecuado para todo tipo de propósito.Los consumidores deben determinar, en gran medida por su cuenta, si un producto software es fiable, seguro y eficaz. La fiabilidad y la credibilidad de las mediciones y los métodos de prueba son un paso importante para asegurar la calidad de los sistemas software, y para promocionar la competitividad en el mercado de la tecnología de la información. Muchas organizaciones en el mundo están trabajando para alcanzar este objetivo (Wakid *et al*., 1999).

1.3 Objeto de las pruebas del software

Normalmente, la prueba del software se realiza a diferentes niveles en el transcurso de los procesos de desarrollo y mantenimiento, lo que significa que su objeto puede cambiar desde un módulo, un grupo de módulos-relacionados por propósito, uso, comportamiento o estructura-,hasta un sistema completo. De acuerdo con esto, conceptualmente se pueden distinguir tres niveles de prueba: de unidad, de integración y de sistema. No se conoce un modelo que determine el proceso implícito, ni como asumir que alguno de estos niveles tenga mayor importancia que los otros (Endres, 1975; Beizer, 1990; Pfleeger, 2001).

1.3.1 Pruebas de unidad

Es un proceso de prueba individual a los sub-programas, sub-rutinas o procedimientos en un programa, también llamados módulos. Es decir, en lugar de una prueba inicial al programa como un todo, la prueba se enfoca primero en pequeños bloques de la construcción del programa. Las razones para esto son: (1) esta prueba es una forma de gestionar el conjunto de elementos de prueba, debido a que la atención se centra inicialmente en unidades pequeñas del programa,(2) facilita la tarea de depuración, porque los errores que se detectan existen en un determinado móduloy (3) introduce un paralelo entre el proceso de prueba del programa y la oportunidad de probar varios módulos simultáneamente (Beizer, 1990).

El objetivo de las pruebas de unidad es comparar la función de un módulo con alguna especificación funcional o interfaz definida en él. Para enfatizar el objetivo de todos los procesos de prueba suintención no es demostrar que el módulo se ajusta a las especificaciones, sino mostrar que las contradice.Normalmente, las pruebas de unidad se realizan con acceso al código fuente y con el soporte de herramientas de depuración, ya veces implican a los desarrolladores (Perry, 2000).

1.3.2 Pruebas de integración

Luego de realizar las pruebas a cada uno de los componentes individuales y probar que funcionan«correctamente» y que satisfacen la especificación, se combinan en un sistema activo. Esta integración se planea ycoordina para que al detectar un errortambién se pueda conocer su origen. Para grandes sistemas es posible que algunos componentes estén en la fase de codificación, otros en la fase de prueba unitaria, y otros en la fase de pruebas de integración, por lo tanto, la estrategia de prueba explicapor qué y cómo se combinan los componentes para probar el sistema activo. Esta estrategia no solo afecta la sincronización de la integración y el orden de codificación, sino también el costo y el detalle de la comprobación. En estas pruebas se ve al sistema como una jerarquía de componentes, en la que cada uno pertenece a una capa del diseño, ylos casos de prueba se pueden ejecutaraplicando un métododescendente-desde lo general-, o hacerlo con un criterio ascendente-desde lo particular-. También se puede utilizar una combinación de ambos enfoques. Las pruebas de integración tienen dos grandes objetivos: (1) detectar los errores que ocurren en las interfaces de unidades y (2) ensamblar las unidades individuales en subsistemas de trabajo paraobtener el sistema completo que se va a probar.

En las pruebas de unidad los probadores tratan de detectar los errores que se relacionan con la funcionalidad y la estructura de la unidad, por lo que existen pruebas tan simples como probar la interacción entre las interfaces de unidad con los controladores. Sin embargo, es más adecuado probar las interfaces en las pruebas de integración, en las que cada unidad se conecta con una

implementación, e inclusive se pueden interconectar. Como consecuencia de este ensamble o proceso de integración, los subsistemas software y el sistema completose ensamblan para ejecutar las pruebas de integración (Burnstein, 2003).

1.3.3 Pruebas del sistema

La prueba del sistema es el nivel más cercano a la experiencia cotidiana. Se prueban muchas cosas: un auto antes de comprarlo, una red en línea de servicio antes de la suscripción, y así sucesivamente. Un patrón común en estas pruebas es evaluar un producto en términos de las expectativas, y no con respecto a un pliego de condiciones o a una norma. En consecuencia, el objetivo no es encontrar errores sino demostrar su rendimiento. Por esto se tiene la tendencia aabordar la prueba del sistema desde un punto de vista funcional en lugar de uno estructural. En la práctica, y debido a que es tan intuitivamente familiar, la prueba del sistema tiende a ser menos formal de lo que debería ser, lo que se agravapor la reducción del intervalo de tiempo invertido en ella,y a que usualmente es antes de que finalice el plazo de entrega (Raymond, 2001).

La prueba del sistema se ocupa del comportamiento del sistema completo, por lo que la mayoría de loserrores funcionales se deberían haber identificado antes,en las pruebas de unidad o de integración. Normalmente se considera como la más apropiada para comparar el sistema con sus requisitos no-funcionales, como seguridad, velocidad, exactitud y confiabilidad. También se evalúan en este nivel las interconexiones externas con otras aplicaciones, utilidades, dispositivos hardware o con el sistema operativo.

1.4 Objetivos de las pruebas del software

Las pruebas se aplicancon un mismo objetivo en una amplia gama de proyectos desarrollados en un gran número de industrias: reducir el riesgo de gastos no planificados en el desarrollo o, peor aún, el riesgo de fracaso del proyecto. Estos riesgosse pueden cuantificar como la pérdida de algún tipo de bien tangible-como ingresos o clientes-.Con el fin de conocer el tamaño del riesgo y la probabilidad de que se produzca, es necesario realizar una valoración, queconsiste en una serie de preguntas estructuradas del tipo «*qué sucedería si*», y que sondean las causas más probables del fracaso del desarrollo en función de su tipo y del tipo de negocio que debe apoyar. Esta valoración del riesgo tiene cuatro objetivos interrelacionados en las pruebas (Everett y McLeod, 2007):

1. Identificar la magnitud y las fuentes de desarrollo de riesgos que se puedan reducir con las pruebas. Si en el proyecto los factores de riesgo son conocidos, cuantificables y controlables, es factible que la prueba pueda reducir la probabilidad de que se generen riesgos. Las pruebas pueden ser totalmente controladas,planificadas y completadas, pero si los factores de riesgo no están bajo el control del proyecto o son difusos -no se conocen bien o son cualitativos-, la prueba no tiene una buena oportunidad de reducir la aparición del riesgo.

2. Realizar pruebas para reducir riesgos identificados. La planificación de pruebas hace hincapié en las áreas de riesgo, a fin de que el mayor porcentaje de la prueba y el esfuerzo -pruebas positivas y negativas-se dedique a la reducción de esos riesgos. Rara vez se ejecutan pruebas para eliminar completamente un riesgo, porque siempre hay más situaciones para probar que tiempo o recursos para completarlas, y debido a que hasta ahora probar cien por ciento el software es una expectativa poco realista de negocios.

3. Saber cuándo se ha completado una prueba. Conociendo lo anterior, el probador debe aplicar algún tipo de prioridades para determinar cuándo detener la prueba. Esa determinación debería comenzar con la prueba de los elementos positivos en el plan de pruebas. El probador debe completarlas las positivas que validan todos los requisitos del desarrollo, de otro modo estaría introduciendo riesgos en el proceso.

4, Gestionar las pruebas como un estándar en el desarrollo del proyecto. Esto garantiza que no se descuiden o se pierdan las pruebas. Pueden ser planeadas y completadas con la misma confianza de cualquier proyecto profesional, hasta alcanzar sus objetivos. El riesgo de esto es que la realización de las pruebas sea un recurso limitado, por lo que, cuando todos los posibles probadores están programados para una inminente prueba del proyecto, no se puede programarla realización de nuevas pruebas hasta que se encuentren más probadores cualificados.

Otros objetivos importantes de las pruebas incluyen mediciones de confiabilidad, evaluación de la facilidad de uso y aceptación, para los cuales se utilizarán métodos diferentes. Se debe tener en cuenta que los objetivos de las pruebas varían con el objeto de las mismas.

1.4.1 Pruebas de aceptación o Validación

Estas pruebas las aplica la organización que solicitó el sistema y tienen como propósito lograr una revisión final al producto, quea menudo es la misma validacióndel sistema. El sistema se prueba en su ambiente real por un período extenso de tiempo; cuando se termina la prueba se toma la decisión de aceptar, rechazar o condicionar el producto. Este tipo de prueba a veces es conocida como prueba *Alfa*. Si no existe un cliente en particular que haya solicitado el sistema, por ejemplo en el caso de un producto de software de venta al público, comúnmente se aplica una prueba *Beta*. Esto significa que antes de enviar el producto, se prueba por clientes seleccionados que utilizan el sistema y reportan los errores encontrados (Weitzenfeld, 2004).

Las pruebas de aceptación comparan el comportamiento del sistema con los requisitos, sin importar cómo estén especificados. El cliente realiza o especifica tareas típicas para comprobar que se satisfacen sus requisitos o para comprobar que la organización los ha identificado para el mercado al que se destina el software. Esta actividad puede incluir o no a los desarrolladores del sistema.

1.4.2 Pruebas de instalación

La fase de instalación plantea dos dificultades para el equipo de pruebas: (1) es un proceso separado del desarrollo de la aplicación y su objetivo no es satisfacer las necesidades de los usuarios sino de colocar en producción una aplicación completa y probada -en muchos casos estas pruebas las realiza un grupo diferente al queaplicó las demás pruebas-y (2) normalmente ocurre en un lapso de tiempo muy corto; por lo tanto, estas pruebas se deben planificary ejecutarde forma que sean significativas y útiles para el proceso de instalación (Perry, 2000).

Los resultados de las pruebas que no están disponibles hasta varias horas o días después de la instalación no valen la pena, porque dada su importancia es conveniente que los resultados estén disponibles antes de la finalización de la instalación. En muchos casos esto significa que los posibleshallazgos de la prueba se deben determinarantes de iniciarla, por lo que se pueden ver como pruebas del sistema realizadas en relación con los requisitos de la configuración de hardware, además, los procedimientos para la instalación tambiénse pueden verificar.

1.4.3 Pruebas Alfa y Beta

La mayoría de desarrolladores de productos software llevan a cabo un proceso denominado pruebas *Alfay Beta* para descubrir errores que al parecer solo el usuario final podría descubrir. La primera la lleva a cabo el cliente en el lugar de desarrollo, para lo cual utiliza el software de forma natural, con el desarrollador como observador, y registra los errores y los problemas de uso todo en un entorno controlado. La prueba Betala aplican los usuarios finales en el lugar de trabajo de los clientes. A diferencia de las primeras, en estas normalmente el desarrollador no está presente. La prueba Betaes una aplicación «en vivo» del software en un entorno que no puede ser controlado por el desarrollador. El cliente registra todos los problemas, reales o imaginarios, que encuentra durante la prueba, e informa a intervalos regulares al desarrollador. Como resultado de los informes, el desarrollador lleva a cabo modificaciones y prepara una versión del producto para todos los clientesa la vez (Pressman, 2004).

1.4.4 Pruebas de conformidad, pruebas funcionales, pruebas de corrección

Este tipo de prueba se basa en la ejecución, revisión y retroalimentación de las especificaciones diseñadas para el software, y se aplica mediante el diseño de modelos que buscan evaluar cada una de las opciones del paquete informático. Su objetivo es validar si el comportamiento observado del software cumple o no con esas especificaciones, para lo que se toma el punto de vista del usuario (Beizer, 1990). Las funciones se prueban ingresando valores de entradas y examinando y comparando las salidas con los valores esperados, pero en pocas ocasiones se considera la estructura del código (Kaner *et al.*, 1999). En la realización de estas pruebasse analiza la especificación de requisitos para derivar los casos de prueba. Técnicas como partición de equivalencia, análisis del

valor límite, grafo causa-efecto, casos de prueba desde los casos de uso, conjetura de errores, son especialmente pertinentes. Se deben considerar las condiciones inválidas e inesperadas de entrada, y tener en cuenta que la definición del resultado esperado es una parte vital del caso de prueba. El propósito es mostrar discrepancias con la especificación y no la de demostrar que el programa cumple con la misma (Myers, 2004).

1.4.5 Materialización de la confiabilidad y la evaluación

La confiablidad es una función que depende de las entradas que el sistema recibe y del uso que se hace de él, y se define como la probabilidad de que el software cause un fallo del sistema durante un tiempo determinado y bajo condiciones específicas. Las entradas al sistema estipulan si los errores, en caso de que existan, se pueden detectar (ANSI/IEEE, 1983). Al ayudar a identificar erroreslas pruebas son un medio que permite mejorar la confiabilidad, pero generar casos de prueba aleatorios siguiendo el perfil de operaciones puede derivaraproximaciones estadísticas de confiabilidad. Cuando se aplica algún modelo para potenciar la confiabilidad, como el de Análisis de Causa Raíz (RCA por sus siglas en inglés), ambos objetivos se pueden alcanzar al mismo tiempo.

1.4.6 Pruebas de regresión

El software cambia cada vez que se añade un nuevo módulo como parte de una prueba de integración, se establecen nuevos caminos de flujo de datos, ocurren nuevas entradas y salidas, o se invoca una nueva lógica de control. Estos cambios pueden causar problemas en funciones que antes trabajaban perfectamente. En el contexto de una estrategia de prueba de integración, correr la prueba de regresión es volver a ejecutar un subconjunto de casos de prueba, aplicados con anterioridad, para asegurar de que los cambios no han propagado efectos colaterales no deseados (Pressman, 2004). En la práctica, la idea es demostrar que un software, que previamente pasó un conjunto de pruebas, aún lo pase (IEEE, 2002). El proceso es cualquier repetición de pruebas que tiene como objetivo demostrar que el comportamiento del software no ha cambiado, excepto en aquellos aspectos en los que se haya requerido. Se debe llegar a un compromiso entre la realización de pruebas de regresión, cada vez que se hace un cambio, y los medios disponibles para realizarlas. Estas pruebas se pueden realizar en cada uno de los niveles de prueba descritos y son válidas tanto para pruebas funcionales comopara no funcionales (Beizer, 1990).

1.4.7 Pruebas de rendimiento

Los requisitos se clasificanen funcionales y no funcionales;los primeros describen las funciones que el software debe desarrollar, y los segundos los niveles de calidad esperados en el producto. El objetivo de las pruebas de rendimiento es probar si el software cumple con los requisitos de funcionamiento. Los probadores las aprovechan para determinar si existe algún factor de hardware o de software que incida en el rendimiento del sistema, para optimizar la asignación de los recursos y proyectar los niveles de rendimiento del sistema, lo que les permite planificar futuras versiones.

Los requisitos se documentan de dos formas: en palabras de la terminología del cliente, y como un conjunto de requisitos de hardware y software que utilizan los desarrolladores. La prueba funcional compara el sistema en construcción con las funciones descritas en la especificación, y la prueba de rendimiento con el resto de requisitos del software y el hardware. Cuando la prueba de rendimiento se lleva a cabo exitosamente, en el ambiente real de trabajo del cliente,se valida el sistema, y cuando se completa los desarrolladores tienen la certeza de que el sistema funciona de acuerdo con su comprensión de la especificación. El próximo pa¬so es dialogar con el cliente para tener la certeza de que el sistema trabaja de acuerdo con sus expectativas.

Los resultados de las pruebas de rendimiento son cuantificables porqueal finalizar el probador sabe, por ejemplo, el número de ciclos de procesador utilizado, el tiempo de respuesta en cuestión de segundos o minutos, y el número real de las transacciones procesadas por período de tiempo, y estos datos pueden ser evaluados con respecto a los objetivos de los requisitos.

1.4.8 Pruebas de deterioro

Las modificaciones que se hacen al software para corregirleerrores o para introducir nuevas funciones originan nuevos errores,y provocanel llamado diente de sierra. Cuando el software se deteriora, es decir, cuando no cumple con las especificaciones para las que fue creado, cada modificación genera nuevos errores que son difíciles de corregir, y este proceso se repite hasta que es necesario reemplazar completamente el software. La mayoría de productos sufren deterioro normal debido principalmente a los errores descubiertos, cambios en el contexto o cambiosen los requisitos de los usuarios, por lo que es necesario hacerles mantenimiento. Asi mismo, el uso de Internet como plataforma de desarrollo está dando lugar a otro tipo de deterioro, como los causados por los virus y los ataques a la seguridad, por lo que estas pruebas deben hacer funcionar el software a su máxima capacidad de diseño, y en ocasiones inclusive por encima de ella.

1.4.9 Pruebas de continuidad

Su objetivo es ampliar el campo de comparación entre los resultados de las pruebas aplicadas a dos o más versiones de un mismo producto. La idea de estas pruebas es tener métricas estadísticas que sustenten los resultados de varias pruebas de un producto software -en varios momentos y en varias versiones- para realizar análisis de vida útil, rendimiento o desgaste.

1.4.10 Pruebas de recuperación

Es una actividad orientada a probar qué tan bien se recupera una aplicación de un fracaso, un error de hardware u otros problemas similares. Las pruebas de recuperación son errores forzados del software en una variedad de formas, para verificar que su recuperación se realiza correctamente; su objetivo es verificar la capacidad del software para recuperarse después de un

«desastre». No se deben confundir con las pruebas de fiabilidad, porque estas intentan descubrir el punto concreto en el que falló el sistema, pero están relacionadas, y de acuerdo con los atributos de calidad propuestos por los estándares corresponden a la característica de confiabilidad. Algunos ejemplos de pruebas de recuperaciónson (Perry, 2000):

1. Mientras que una aplicación se ejecuta, reiniciar el equipo y después comprobar la validez de integridad de los datos de la aplicación.
2. Mientras que una aplicación recibe datos de una red, desconectar el cable y después de algún tiempo conectarlo nuevamente y analizar la capacidad de la aplicación para continuar recibiendo los datos desde el punto en el que desapareció la conexión.
3. Mientras el navegador tiene cierto número de sesiones abiertas reiniciar el sistema y después comprobarsi es capaz de recuperarlas todas.

1.4.11 Pruebas de configuración

Estas pruebasverifican la operación del sistema en diferentes configuraciones de hardware y software, para lo que, por ejemplo, se valida y se verifica si la aplicación funciona bien en diferentes estaciones de trabajo. Estas pruebas las realiza el implementador debido al nivel de detalle requerido y al conocimiento sobre las fuentes creadas. Se enfocan en evaluar las diferentes variaciones de una aplicación integrada contra sus requisitos de configuración. La meta es hacer que la aplicación falle en cumplir sus requisitos de configuración, de forma que los errores escondidos se identifiquen, analicen, corrijan y prevengan en el futuro.

Para determinar si los componentes de la aplicación funcionan correctamente cuando se unen, se deben probar partes combinadas, que pueden ser módulos referentes a un caso de uso, además, se debe probar si se cumplen las especificaciones globales de requisitosy si el producto cumple con todos los requisitos y los criterios de aceptación. Es necesario incluir estas pruebas después de realizar mejoras o correcciones de errores.

1.4.12 Pruebas de facilidad de uso

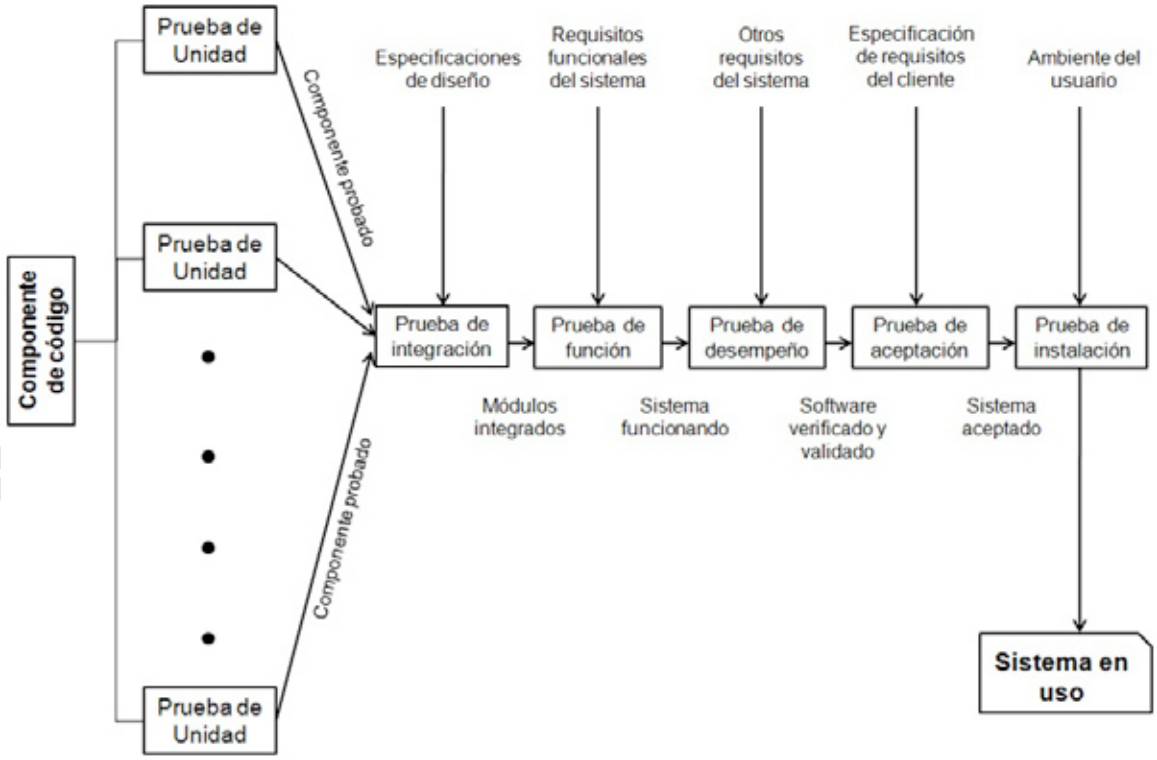
Evalúan la facilidad de uso de una aplicación, ya sea mediante observación directa de las personas que la utilizan, mediante encuestas de la facilidad de uso oa través de pruebas Beta. El objetivo de estas pruebas es asegurar que una aplicación sea fácil de comprender y de navegar (Perry, 2000); es una técnica utilizada para evaluar un producto mediante pruebas de usuarios. Este proceso evalúa lo fácil que le resultausar y aprender a utilizar el software, incluyendo la documentación, la efectividad de las funciones del software para soportar las tareas y la habilidad para recuperarse de errores provocados por el usuario.Puede verse como una práctica insustituible de usabilidad,

porque da una idea directa de la manera como los usuarios utilizan el sistema, en contraste con los métodos de inspección de facilidad de uso, donde se utilizan diferentes métodos para evaluar una interfaz de usuariopero sin su participación (Nielsen, 1994). Las pruebas de facilidad de uso se centran en la medición de la capacidad de un productopara cumplir su finalidad. Ejemplos de productos a los que comúnmente se aplican estas pruebas son: los sitios Web o aplicaciones Web, interfaces, documentos o dispositivos.

1.4.13 Desarrollo dirigido por pruebas

No es una técnica en sí misma, pero promociona el uso de pruebas como una parte subordinada al documento de especificación de requisitos, en vez de una comprobación independiente de que el software los implementa correctamente. Como una extensión a la realización de pruebas de software, este concepto indica que «alguien» tiene que escribir las pruebas del código a partir de su especificación y antes que se implemente. Esto se debe a que su objetivo es dual: (1) escribir pruebas para un código que no existe, debido a que es conveniente realizarla en la fase de diseño del software que se va a construir y (2) que las pruebas realizadas obedezcan a una especificación. Por lo tanto, es posible identificar en qué parte del software se implementó la especificación requerida(Beck, 2003).El proceso del desarrollo dirigido por pruebasse representa en la Figura4.

Figura 4. Pasos del desarrollo dirigido por pruebas (Pfleeger, 2001)



5. Propuestas para aplicar técnicas de prueba

Los diferentes tipos y niveles de pruebas requieren técnicas para aplicar su metodología, ya quea su vez las técnicas de pruebanecesitan propuestas de aplicación para llevarlas a la práctica. En las diferentes épocas de la historia de la Verificación del software, se propusieron técnicas de prueba que dieron origen a investigaciones que culminaron con diferentes propuestas para aplicarlas. Cabe recordar lo sucedido con las pruebas funcionales: surgieron en los 80 como respuesta a las necesidades de incrementar la calidad del software y a la inquietud de la época por hacer de las pruebas un proceso continuo y paralelo al ciclo de vida de los proyectos de desarrollo. Rápidamente se estructuraron técnicas que permitieron aplicar esa metodología, pero hacía falta cómo ponerlas en práctica, por lo que algunos investigadores se dieron a la tarea de realizar proyectos experimentales que condujeron a las primeras propuestas de aplicación de esas técnicas en las pruebas funcionales. Ejemplos de entonces se pueden citar las técnicas *Equivalence Partitioning y Boundary Value Testing*, propuestas por Bohem (1979), y para las cuales, en menos de un año, se diseñaron varias propuestas de aplicación (Juristo et al., 2004).

Mientras que una técnica de prueba se utiliza para diseñar y seleccionar los casos de prueba que se aplicarán, de acuerdo con el tipo de prueba determinado, una propuesta de prueba resulta de procesos de investigación que buscan la mejor forma de aplicar esa técnica. Debido a que cada técnica debe respetar unas reglas a través de las cuales es posible conocer si el diseño de la prueba cumple o no con la metodología, requiere propuestas de aplicación que se puedan programar y automatizar en la práctica continua del proceso de la prueba (Myers, 2004).

Referencias

Abran, A., Bourque, P., Dupuis, R. and Moore, J. W. (2004). *Guide to the Software Engineering Body of Knowledge –SWEBOK*. IEEE Press.

Akiyama, K. (1971). An example of software system debugging. *Proceedings of the 1971 IFIP Congress.* Amsterdam, Netherlands, pp. 353-358.

Alberts, D. S. (1976).The economics of software quality assurance.*Proceedings of the 1976 National Computer Conference and exposition*. New York, USA.

ANSI/IEEE (1983).Glossary of Software Engineering Terminology Standard 729-1983. IEEE Press.

Arthur, L. J. (1997). *Improving Software Quality: An Insider's Guide to TQM.* John Wiley y Sons.

Bache, R. y Müllerburg, M. (1990). Measures of Testability as a Basis for Quality Assurance.*Software Engineering Journal*. Vol. 5, No. 2, pp. 86-92.

Basili, V. R. y Selby, R. W. (1987). Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*. Vol. 13, No. 12, pp. 1278-1296.

Beck, K. (1999).*Extreme Programming Explained.* Addison-Wesley.

Beck, K. (2003).*Test-Driven Development By Example.* Addison Wesley.

Beizer, B. (1990). *Software Testing Techniques.* Van Nostrand Reinhold.

Beizer, B. (1995). *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley.

Bertolino, A. y Marrè, M. (1996). How Many Paths Are Needed for Branch Testing? *Journal of Systems and Software.* Vol. 35, No. 2, pp. 95-106.

Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Professional.

Boehm, B. W. (1975). *The high cost of software: Practical Strategies for Developing Large Software Systems*. Addison-Wesley.

Bohem, B. W. (1979). Guidelines for verifying and validating software requirements and design specifications. *European conference on applied information technology of the international federation for information processing.* London, UK, pp. 711-719.

Brown, J. R. y Lipow, M. (1973). The quantitative measurement of software safety and reliability. TRW Report SDP-1176.

Burnstein, I. (2003). Practical software testing: a process-oriented approach. Springer-Verlag.

Buy U., Orso A. y Pezzè, M. (2000). Automated testing of clases. *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis.* New York, USA, pp. 39–48.

Dijkstra, E. W. (2000). Triggered by a high-school exercise.EWD 1297. Oline: http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1297. PDF [Jan. 2011].

Endres, A. (1975). An analysis of errors and their causes in system programs.*IEEE Transactions on Software Engineering,* Vol. 1, No. 2, pp. 140-149.

Everett, G. D. y McLeod, R. (2007). Software Testing: Testing Across the Entire Software Development Life Cycle. John Wiley y Sons.

Firesmith, D. G. (1993).Testing Object-Oriented Software. *Software Engineering Strategies.* Vol. 1, No. 5, pp. 15-35.

Ford, G. y Gibbs, N. E. (1996).A Mature Profession of Software Engineering.Software Engineering Institute, Carnegie Mellon University. Pittsburgh, Tech. report CMU/SEI-96-TR-004.

Frankl, P. G. et al. (1998). Evaluating Testing Methods by Delivered Reliability. *IEEE Transactions on Software Engineering.* Vol. 24, No. 8, pp. 586-601.

Frankl, P. G. y Weyuker, E. J. (1993). An analytical comparison of the fault-detecting ability of data flow testing techniques.*Proceedings of the 15th International Conference on Software Engineering.* New York, USA, pp. 415-424.

Glass, R. L. (1981). Persistent Software Errors. *IEEE Transactions on Software Engineering.* Vol. 7, No. 2, pp. 162-168.

Glass, R. L. (1981a). Software Errors and Complexity: An Empirical Investigation. *Communications of ACM.* Vol. 27, No. 1, pp. 42-52.

Goodenough, J. B. y Gerhart, S. L. (1977). Toward a Theory of Testing Data Selection Criteria. *Current Trends in Programming Methodology*. Vol. 2, pp. 44-79.

Goodenough, J. B. (1979). A survey of program testing issues.In P. Wegner Ed. "Research Directions in Software Technology". MIT Press, pp. 316-340.

Grady, R. (1992).Practical Software Metrics for Project Management and Process Improvement. Prentice Hall.

Harrold, M. J. y Rothermel, G. (1994). Performing data flow testing on clases. *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. New York, pp. 154-163.

Harrold, M. J. y Soffa, M. L. (1994). Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*. Vol. 16, No. 2, pp. 175-204.

Hetzel, W. C. (1988). The Complete Guide to Software Testing. Wellesley.

Hollnagel, E. (1993). *The phenotype of erroneous actions.International Journal of Man-Machine Studies*, Vol. 39, No. 1, pp. 1-32.

Howden, W. E. (1976). Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*. Vol. 2, No. 3, pp. 208-215.

Howden, W. E. (1978). A survey of dynamic analysis methods.In E. Miller y W. E. Howden (Eds.) Software Testing and Validation Techniques. *IEEE Press*, pp. 184-206.

Huffman, J. H. (1994). Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach.*Lectures Notes in Computer Science*. Vol. 858, pp. 205-220.

Hutcheson, M. L. (2003). Software testing fundamentals: methods and metrics. John Wiley y Sons.

IEEE (1990). Standard Glossary of Software Engineering Terminology Std 610.12-1990. IEEE Press.

IEEE (1993). Standards Collection for Software EngineeringStd. 1028. IEEE Press.

IEEE (1993a). Classification for Software Anomalies Std. 1044-1993. IEEE Press.

IEEE (1997). Standard for developing software life cycle Processes Std. 1074-1997. IEEE Press.

IEEE (1998). Standard for Software Test Documentation Std. 829-1998. IEEE Press.

IEEE (2002). Standard Glossary of Software Engineering Terminology Std. 610.12. IEEE Press.

IEEE (2005). Dictionary of Measures of the Software Aspects of Dependability Std. 982.1-2005. IEEE Press.

ISO/IEC (1995). Standard for Information Technology Software Life Cycle Processes 12207. IEEE Press.

ISO/IEC (2001). Software engineering: Product quality std ISO/IEC 9126. Geneva: ISO/IEC Press.

Jorgensen, P. C. (2004). Software Testing: A Craftsman's Approach. Washington: CRC Press.

Juristo, N. et al. (2004). Reviewing 25 Years of Testing Technique Experiments. *Empirical Software Engineering.* Vol. 9, No. 1-2, pp. 7-44.

Kaner, C. et al. (1999). Testing Computer Software.John Wiley y Sons.

Kay, R. H. (2007). The role of errors in learning computer software.*Computers y Education.* Vol. 49, No. 2, pp. 441-459.

Knuth, D. E. (1989).The Errors of Tex".*Software Practice and Experience.* Vol. 19, No. 7, pp. 607-685.

Lazonder, A. W. y Van Der Meij, H. (1995). Error-information in tutorial documentation: supporting user's errors to facilitate initial skill learning. *International Journal of Computer Studies.* Vol. 42, No. 2, pp. 185-206.

Leszaka, M. et al. (2002). Classification and evaluation of defects in a project retrospective.*The Journal of Systems and Software.* Vol. 61, No. 3, pp. 173-187.

Levendel, Y. H. (1990). Reliability analysis of large software systems: Defect data modeling. *IEEE Transactions on Software Engineering.* Vol. 16, No. 2, pp. 141-152.

Lewis, W. E. (2009). Software testing and continuous quality improvement.Auerbach Publications.

Marick, B. (1990). A Survey of Test Effectiveness and Cost Studies.Technical Report UIUCDCS-R-90-1652, University of Illinois.

McGregor, J. D. y Sykes, D. D. (2005). A practical guide to testing object-oriented software. Addison-Wesley.

Miller, E. F. (1979). Program testing: an overview for managers. *Software testing*. Vol. 12, No. 3, pp. 187-200.

Moore, J. W. (1994). An integrated collection of software engineering standards. *IEEE software*. Vol. 16, No. 6, pp. 51-57.

Morell, L. J. (1990). A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*. Vol. 16, No. 8, pp. 844-857.

Musa, J. D. y Widmaier, J. (1998). Software Reliability Engineered Testing. McGraw Hill.

Musa, J. D. (2004). More Reliable Software Faster and Cheaper: Software Reliability Engineering. AuthorHouse.

Myers, G. J. (2004). The Art of Software Testing.John Wiley y Sons.

Nebut, C. et al. (2004). A Requirement-Based Approach to Test Product Families. *Lecture Notes in Computer Science*. No. 3014, pp. 198-210.

Nielsen, J. (1994). Usability Engineering. Morgan Kaufmann.

Offutt, A. J. (1994). A practical system for mutation testing: help for the common programmer. *Proceedings of the IEEE International Test Conference on TEST: The Next 25 Years*, Washington, USA, pp. 824-830.

Ostrand, T. J. y Weyuker, E. J. (1984). Collecting and Categorizing Software Error Data in an Industrial Environment. *Journal of Systems and Software*. Vol. 4, No. 4, pp. 289-300.

Overbeck, J. (1993).Testing Object-Oriented Software: State of the Art and Research Directions. *First EuroSTAR Conference. London*, UK, pp. 5-25.

Pande, H. D. et al.(1994). Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*. Vol. 20, No. 5, pp. 385-403.

Patton, R. (2000). Software testing.Sams Publishing.

Perry, W. (1995). Effective Methods for Software Testing.John Wiley y Sons.

Perry, W. (2000). Effective methods for Software Testing.Wiley.

Perry, W. E. (2006). Effective methods for software testing.John Wiley y Sons.

Pezzè, M. y Young M. (2006). Software Test and Analysis: Process, Principles and Techniques. John Wiley y Sons.

Pfleeger, S. L. (2001). Software Engineering: Theory and Practice. Prentice-Hall.

Pressman, R. (2004). Software engineering. A Practitioner's Approach. McGraw-Hill.

Purchase, J.A y Winder R. L. (1991).Debugging Tools for Object-Oriented Programming. Object-Oriented Programming. Vol. 4, No. 3, pp. 10-27.

Radatz, J. W. (1981). Analysis of IVyV data. Rome Air Development Center Report RADC-TR-81-145.

Raymond, S. E. (2001). The cathedral and the bazaar: musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly Media.

Richardson, D. J. y Thompson, M. C. (1988). The RELAY model of error detection and its application. *2nd Workshop on Software Testing, Verification and Analysis.* New York, USA, pp. 223-230.

Schulmeyer, G. G. y McManus, J. I. (1999). Handbook of Software Quality Assurance.Prentice Hall.

Schwaber, K. et al. (2001). Agile Software Development with Scrum.Prentice Hall.

Shooman, M. L. y Bolsky, M. I. (1975). Types, distributions and test and correction times for programming errors. *Proceedings of the international conference on Reliable software*. Los Angeles, USA.

Sommerville, I (2006). Software engineering. Prentice Hall.

Souter, A. L. y Pollock, L. L. (2003). The construction of contextual def-use associations for objectoriented systems. *IEEE Transactions on Software Engineering.* Vol. 29, No. 11, pp. 1005-1018.

Starr, P. (1982). The Social Transformation of American Medicine.Basic Books.

Suryn, W. y Abran, A. (2003). ISO/IEC SQuaRE. The second generation of standards for software product quality. *IASTED 2003 - SEA 2003.* Marina del Rey, USA.

Thayer, T. A. (1976). Software reliability study. Report RADC-TR77-216, Rome Air Development Center.

Tuya, G. P. J. et al. (2007). Técnicas cuantitativas para la gestión en la ingeniería del software. Netbiblo.

Virvou, M. (1999). Automatic reasoning and help about human errors in using an operating system. *Interacting with Computers.* Vol. 11, No. 5, pp. 545-573.

Voas, J. (1990). A Dynamic Failure Model for Propagation and Infection Analysis on Computer Programs. Ph.D. Thesis. Virginia: College of William and Mary.

Voas, J. M. y Miller, K. W. (1995). Software Testability: The New Verification. *IEEE Software.* Vol. 12, No. 3, pp. 17-28.

Vogel P. A. (1993). An integrated general propose automated teat environment. *Proceedings of the 1993 International Symposium on Software Testing and Analysis.* Cambridge, UK, pp. 61-69.

Wakid S., Kuhn D. R. y Wallace, D. R. (1999). Toward Credible IT Testing and Certification. *IEEE Software.* Vol. 16, No. 4, pp. 39-47.

Weinberg, G. M. (1998). The Psychology of Computer Programming. Dorset House Publishing Co.

Weiss, D. M. y Basili, V. R. (1985). Evaluating software development by analysis of changes: some data from the software engineering laboratory. *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 2, pp. 157-168.

Weiss, D. M. (1979). Evaluating Software Development from Error Analysis: The Data from the Architecture Research Facility.*Journal of Systems and Software*. Vol. 1, pp. 57-70.

Weitzenfeld, A. (2004). Object Oriented Software Engineering with UML, Java and Internet. Thompson.

Weyuker, E. J. y Ostrand, T. J. (1980). Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering.* Vol. 6, No. 3, pp. 236-246.

Weyuker, E. J. (1983). Assessing Test Data Adequacy through Program Inference. *ACM Transactions On Programming Languages and Systems.* Vol. 5, No. 4, pp. 641-655.

Weyuker, E. J. et al. (1991). Comparison of Program Test Strategies. *Proccedings Symposium on Testing, Analysis and Verification.* Victoria, Canada.

Whittaker, J. A. (2000). What is Software Testing? And Why Is It So Hard? *IEEE Software.* Vol. 17, No. 1, pp. 70-79.

Wolverton, R. W. (1974). The cost of developing large-scale software. *IEEE Transactions on Computers.* Vol. 23, No. 6, pp. 615-636.

Zhu, H. et al. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys.* Vol. 29, No. 4, pp. 366-427.

Edgar Serna

Científico computacional teórico, con más de 10 años de experiencia en la industria como líder de proyectos en Sistemas de Información y como arquitecto de Software, y profesor universitario e investigador con más de 20 años de trayectoria. Sus áreas de investigación son la Lógica, la Ingeniería de Software, las Ciencias Computacionales, los Métodos Formales y las Matemáticas en la Computación, alrededor de las cuales ha hecho varias publicaciones, y participado con ponencias y conferencias en eventos nacionales e internacionales.

Investigación Científica

Las fuentes tipográficas empleadas son *Californian FB* para texto corrido
y *Caviar Dreams* para títulos.

En este libro se describen conceptos específicos para examinar los procesos de verificación del software, desde la perspectiva del análisis a varias propuestas que aplican la técnica Casos de prueba desde los casos de uso, e ilustra la estructura de un importante concepto en las pruebas del software: el esfuerzo de las pruebas se debe integrar linealmente al proceso de desarrollo del producto, desde las fases iniciales del ciclo de vida, para alcanzar logros eficientes y efectivos. Aislar el esfuerzo de la prueba en una caja negra para el final de ese ciclo es un error común y se debe evitar, porque se puede incurrir en un incremento en el tiempo de entrega y de los costos finales.

El libro es el resultado de un proceso de investigación en el cual se evaluaron las propuestas recientes para diseñar casos de prueba para las pruebas funcionales del software. De la muestra inicial, se seleccionaron nueve, que posteriormente se aplicaron en un proceso experimental para validar, verificar y comparar la eficiencia y eficacia de los casos de prueba que estructuran y generan. Luego, se tomó los resultados para diseñar una nueva propuesta metodológica aprovechando las mejores prácticas de cada una. La nueva propuesta se describe y aplica a un estudio de caso, mediante el cual se realiza la prueba funcional del software a través de un proceso de verificación paralelo a la Ingeniería de Requisitos.

This book discusses specific concepts to evaluate software verification process from the perspective of the analysis of various technical approaches using Test cases from use cases. It also illustrates the structure of an important concept in software testing: the effort of these tests should be integrated into the product development process, from the initial stages of the life cycle in order to reach efficient and effective achievements. Isolating the test effort in a black box until the end of that cycle is a common mistake that should be avoided since it can lead to an increase in delivery time and final costs.

The book is the result of a research process in which the recent proposals used to design test cases for functional testing of software were evaluated. Out of the initial sample, nine proposals were selected and subsequently applied in an experimental process to validate, verify and compare the efficiency and effectiveness of test cases which they structure and generate. Then, the results were applied to design a new methodological approach using the best practices of each proposal. The new proposal is described and applied to a case study, in which the functional software test is performed through a verification process parallel to the requirements analysis.

FONDO EDITORIAL ITM

ISBN 978-958-8743-41-7