Introducción al Testing de Software

Maximiliano Cristiá

Ingeniería de Software 2
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

2026

Índice

1. Verificación y validación – Calidad de software **2**

2. Definición de testing y vocabulario básico **3**

3. El proceso de testing **5**

4. Testing de distintos aspectos de un software **7**

5. Las dos estrategias clásicas de testing **7**

> An algorithm is correct with respect to a specification if it behaves as specified.
>
> *Correctness (computer science)* – Wikipedia.org

> Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
>
> *The Humble Programmer (1972)* – E. W. Dijkstra

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> *Comunicación personal con Peter van Emde Boas (1977)* – Donald Knuth

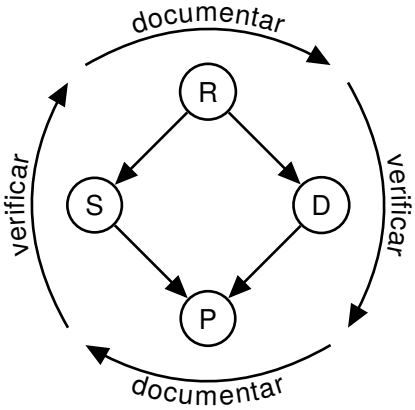

Figura 1: Las cuatro descripciones fundamentales

1. Verificación y validación – Calidad de software

El testing de software pertenece a una actividad o etapa del proceso de producción de software denominada *Verificación y Validación* (V&V)[1]. En la industria no suelen usar el término V&V sino términos como *gestión de la calidad*[2], *aseguramiento de la calidad*[3] o *control de calidad*[4], muchas veces asociados a estándares internacionales[5]. Por otra parte, recuerden que cuando en Ingeniería de Software 1 presentamos las cuatro descripciones fundamentales del software (Figura 1), mencionamos las actividades Documentar y Verificar como actividades que se deberían realizar de forma continua durante el proyecto. Precisamente Verificar es una forma compacta de mencionar o introducir la V&V.

V&V es el nombre genérico dado a las actividades de comprobación que aseguran que el software respeta su especificación y satisface las necesidades de sus usuarios. El sistema debe ser verificado y validado en cada etapa del proceso de desarrollo utilizando los documentos (descripciones) producidos durante las etapas anteriores [1]. En rigor no solo el código debe ser sometido a actividades de V&V sino también todos los subproductos generados durante el desarrollo del software [2]. Por ejemplo, en Ingeniería de Software 1 hemos estudiado cómo verificar una especificación formal utilizando {*log*}. También la arquitectura y el diseño deben ser verificados de forma tal de que exista mayor confianza en que cumplen con los requerimientos del cliente—en particular el equipo de desarrollo debe asegurarse de que la arquitectura podrá incorporar los cambios previstos a bajo costo y que esta habilita otras propiedades que haya solicitado el cliente tales como seguridad, portabilidad, etc.; para más detalles sobre la verificación de arquitecturas se puede consultar [3, Capítulo 11]. Tareas semejantes deben llevarse a cabo para los otros subproductos del desarrollo.

Si bien verificación y validación en su uso cotidiano pueden llegar a ser sinónimos, en Ingeniería de Software tienen significados diferentes y cada uno tiene una definición más o

[1] Wikipedia.org: Verification and validation.

[2] Wikipedia.org: Software quality.

[3] Wikipedia.org: Software quality assurance. Tal vez hayan escuchado que en algunas empresas de software existe un área llamada QA, justamente, por *quality assurance*.

[4] Wikipedia.org: Quality control.

[5] Wikipedia.org: ISO/IEC 9126 (tengan en cuenta que en 2011 ISO/IEC 9126 fue reemplazado por ISO/IEC 25010, aunque los lineamientos generales son los mismos); Capability Maturity Model Integration.

menos precisa.

- *Validación:* ¿estamos construyendo el producto correcto?
- *Verificación:* ¿estamos construyendo el producto correctamente?

En este sentido, la verificación consiste en corroborar que el programa respeta su especificación, mientras que validación significa corroborar que el programa satisface las expectativas del usuario [1]. En otras palabras, la verificación es una actividad desarrollada por ingenieros teniendo en cuenta un modelo del programa y el programa en sí; en tanto que en la validación debe participar el usuario teniendo en cuenta lo que él espera del programa y el programa en sí.

Existen varias técnicas dentro del marco de la V&V desde las más informales (prototipación de requerimientos, revisión de requerimientos, etc.), pasando por las semiformales (el testing es la más conocida pero ciertas técnicas de análisis estático de código se usan frecuentemente), hasta la prueba formal de programas, el cálculo de refinamiento, etc.

Aunque el testing se puede aplicar tanto en verificación como en validación, en este curso nos concentraremos casi exclusivamente en el testing de *programas* como parte de la *verificación*. Es decir que no estudiaremos otras técnicas de verificación, no estudiaremos técnicas de validación y no estudiaremos la V&V de otras descripciones más allá del programa.

2. Definición de testing y vocabulario básico

El testing es una actividad desarrollada para evaluar la calidad del producto, y para mejorarlo al identificar defectos y problemas.

Definición 1 (Testing de software)**.** *El testing de software consiste en la verificación* dinámica *del comportamiento de un programa sobre un conjunto* finito *de casos de prueba, apropiadamente* seleccionados *a partir del dominio de ejecución que usualmente es infinito, en relación con el comportamiento* esperado *[4, 5].*

Las palabras resaltadas en el párrafo anterior corresponden a las características fundamentales del testing. Es una técnica dinámica en el sentido de que el programa se verifica poniéndolo en ejecución de la forma más parecida posible a como ejecutará cuando esté en producción—esto se contrapone a las técnicas estáticas las cuales se basan en analizar el código fuente. El programa se prueba ejecutando solo unos pocos casos de prueba dado que por lo general es física, económica o técnicamente imposible ejecutarlo para todos los valores de entrada posibles—de aquí la frase de Dijkstra. Si uno de los casos de prueba detecta un error[6] el programa es incorrecto, pero si ninguno de los casos de prueba seleccionados encuentra un error no podemos decir que el programa es correcto (perfecto). Esos casos de prueba son elegidos siguiendo alguna regla o criterio de selección. Se determina si un caso de prueba ha detectado un error o no comparando la salida producida con la salida esperada para la entrada correspondiente—la salida esperada debería estar documentada en la especificación del programa.

Las limitaciones antes mencionadas no impiden que el testing se base en técnicas consistentes, sistemáticas y rigurosas (e incluso, como veremos más adelante, formales). Sin embargo, en la práctica industrial, como ocurre con otras áreas de Ingeniería de Software, usualmente se

[6] En el área de testing se distingue con cierta precisión los términos *error* o *defecto*, *falla* y *falta*, aunque por el momento nosotros los consideraremos sinónimos.

considera solo una parte mínima de dichas técnicas tornando a una actividad medianamente eficaz y eficiente en algo fútil y de escaso impacto.

Aunque en la industria de software el testing es la técnica predominante—en aquellos casos minoritarios en que se realiza una actividad seria de V&V—, en el ambiente académico se estudian y desarrollan muchas otras técnicas.

Veamos algunos conceptos básicos de testing. Testear un programa significa ejecutarlo bajo condiciones controladas tales que permitan observar su salida o resultados. El testing se estructura en *casos de prueba* o *casos de test*; los casos de prueba se reúnen en *conjuntos de prueba*. Desde el punto de vista del testing se ve a un programa (o subrutina) como una función que va del producto cartesiano de sus entradas en el producto cartesiano de sus salidas. Es decir:

$$P : ID \rightarrow OD$$

donde *ID* se llama *dominio de entrada* del programa y *OD* es el *dominio de salida*. Normalmente los dominios de entrada y salida son conjuntos de tuplas tipadas cuyos campos identifican a cada una de las variables de entrada o salida del programa, es decir:

$$ID \mathrel{\widehat{=}} [x_1 : X_1, \ldots, x_n : X_n]$$

$$OD \mathrel{\widehat{=}} [y_1 : Y_1, \ldots, y_m : Y_m]$$

De esta forma, un caso de prueba es un elemento, x, del dominio de entrada (es decir $x \in ID$) y testear P con x es simplemente calcular $P(x)$. En el mismo sentido, un conjunto de prueba, por ejemplo T, es un conjunto de casos de prueba *definido por extensión* y testear P con T es calcular $P(x)$ para cada $x \in T$. Es muy importante notar que x es un valor constante, no una variable; es decir, si por ejemplo $ID \mathrel{\widehat{=}} [num : \mathbb{N}]$ entonces un caso de prueba es $num = 5$, otro puede ser $num = 1000$, etc., pero no $num > 24$. Algo como $num > 24$ es una *condición de test* o *especificación de test*.

También es importante remarcar que $x_1, \ldots, x_n$ son las entradas con que se programó el programa (esto incluye archivos, parámetros recibidos, datos leídos desde el entorno, etc.) y no entradas abstractas o generales que no están representadas explícitamente en el código. De la misma forma, $y_1, \ldots, y_m$ son las salidas explícitas o implícitas del programa (esto incluye salidas por cualquier dispositivo, parámetro o valor de retorno, e incluso errores tales como no terminación, `Segmentation fault`, etc.).

De acuerdo a la definición clásica de corrección, un programa es correcto si verifica su especificación. Entonces, en términos de testing, un programa es correcto si ninguno de los casos de prueba seleccionados detecta un error. Precisamente, la presencia de un *error* o *defecto* se demuestra cuando $P(x)$ no satisface la especificación para algún x en *ID*. Una *falla*[7] [2] es el síntoma manifiesto de la presencia de un error. Es decir que un error permanecerá oculto hasta que ocurra una falla causada por aquel. Por ejemplo, si la condición de una sentencia `if` es `x > 0` cuando debería haber sido `x > 1`, entonces hay un error en el programa, que se manifestará (falla) cuando se testee el programa con `x` igual a 1—si por fortuna este caso de prueba es seleccionado—porque en tal caso aparecerá cierto mensaje en la pantalla que para ese valor de `x` no debería haber aparecido. En este sentido el testing trata de incrementar la probabilidad de que los errores en un programa causen fallas, al seleccionar casos de prueba

[7] 'failure', en inglés.

apropiados. Finalmente, una *falta*[8] [2] es un estado intermedio incorrecto al cual se llega durante la ejecución de un programa—por ejemplo se invoca a una subrutina cuando se debería haber invocado a otra, o se asigna un valor a una variable cuando se debería haber asignado otro.

3. El proceso de testing

Es casi imposible, excepto para los programas más pequeños, testear un software como si fuera una única entidad monolítica. Como hemos visto en los capítulos sobre arquitectura y diseño, los grandes sistemas de software están compuestos de subsistemas, módulos y subrutinas. Se sugiere, por lo tanto, que el proceso de testing esté guiado por dicha estructura, como muestra la Figura 2. Más aun, si el proceso sugerido se combina adecuadamente con el proceso de desarrollo, como muestra la Figura 4, entonces se habilita la posibilidad de ir detectando errores de implementación lo más tempranamente posible—lo que a su vez reduce el costo de desarrollo.

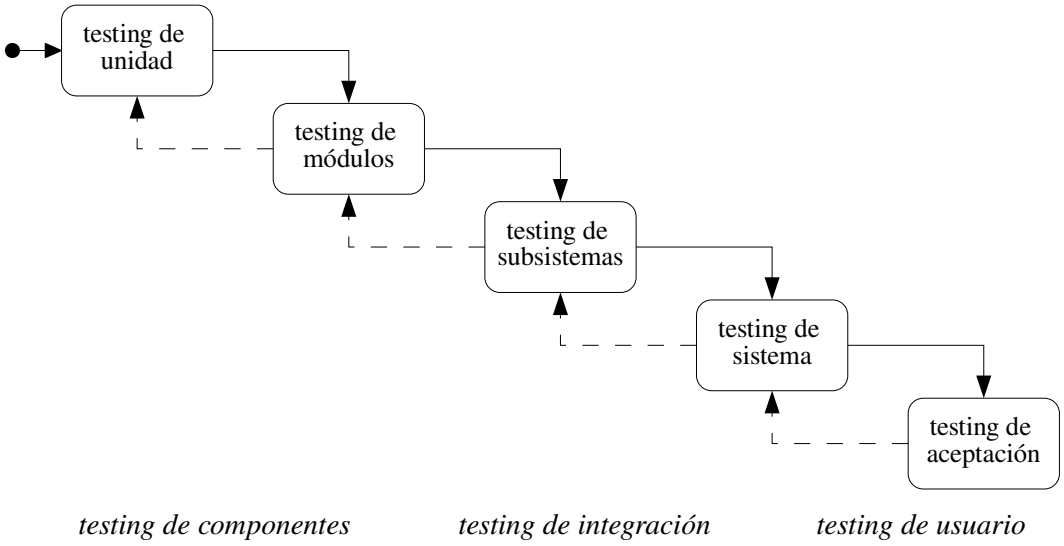

Figura 2: El proceso general de testing [1]

Como sugiere la Figura 2, un sistema complejo suele testearse en varias etapas que por lo general se ejecutan siguiendo una estrategia *bottom-up*, aunque el proceso general es iterativo. Se debe volver a las fases anteriores cada vez que se encuentra un error en la fase que está siendo ejecutada. Por ejemplo, si se encuentra un error durante el testing de un subsistema particular, una vez que aquel haya sido reparado se deberán testear la unidad donde estaba el error y luego el módulo al cual pertenece la unidad reparada. En general, una vez detectado un error se sigue el proceso graficado en la Figura 3. De aquí que a las iteraciones del proceso de la Figura 2 se las llame *re-testing*.

Dado que no hay una definición precisa de subsistema e incluso de unidad, el proceso de testing sugerido debe ser considerado como una guía que debe ser adaptada a cada caso

[8] 'fault', en inglés.

Figura 3: El proceso de *debugging* [1]

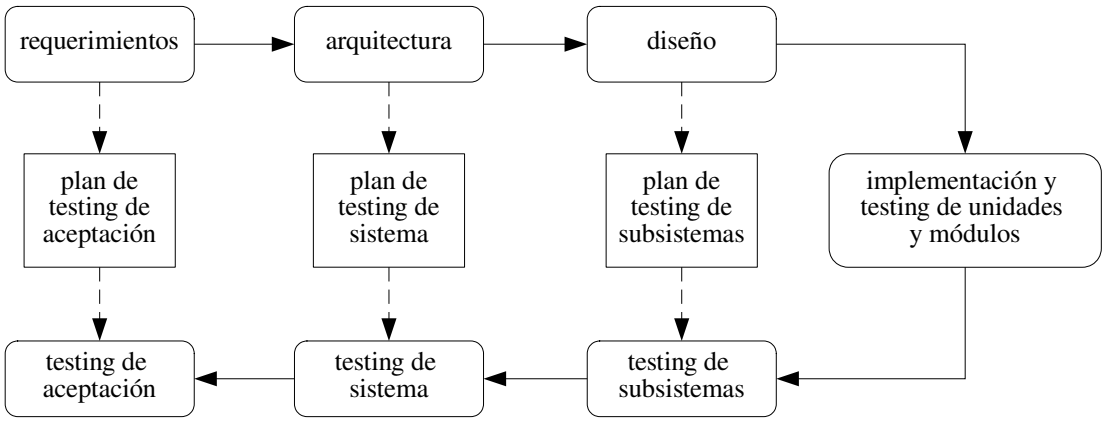

Figura 4: Coordinación entre el proceso de testing y el proceso de desarrollo (adaptado de [1])

específico. El testing de aceptación mencionado en la figura pertenece a la validación de un sistema (y no a la verificación) dado que es el usuario el que usa el sistema en un entorno más o menos real con el fin de comprobar si es una implementación razonable de los requerimientos. En este curso nos concentraremos en el testing de unidad dado que es la base para el resto del proceso.

En la práctica industrial usualmente se entiende que el testing es una actividad que se realiza una vez que los programadores han terminado de codificar; en general es sinónimo de testing de aceptación. Entendido de esta forma el testing se convierte en una actividad costosa e ineficiente desde varios puntos de vista:

- Los testers estarán ociosos durante la mayor parte del proyecto y estarán sobrecargados de trabajo cuando este esté por finalizar.
- Los errores tienden a ser detectados muy tarde.
- Se descubre un gran número de errores cuando el presupuesto se está terminando.
- Los errores tienden a ser detectados por los usuarios y no por el personal de desarrollo—lo que implica un desprestigio para el grupo de desarrollo.

Por lo tanto, se sugiere un proceso de testing mejor imbricado con el proceso general de desarrollo, como se muestra en la Figura 4. En cuanto el equipo de desarrollo cuenta con un documento de requerimientos más o menos estable, los testers pueden comenzar a definir casos de prueba para el testing de aceptación, dado que este se basa en validar los requerimientos. De la misma forma, tan pronto como se ha definido la arquitectura o el diseño del sistema, los testers pueden usar dicha documentación para calcular casos de prueba para el testing de

subsistemas y del sistema. Incluso teniendo un diseño más o menos detallado—descripto con la documentación que hemos analizado en capítulos anteriores—y una especificación (formal o informal) de cada unidad, es perfectamente posible calcular casos de prueba de unidad mientras los programadores hacen su trabajo. De esta forma se reduce notablemente el impacto de algunos de los problemas enumerados más arriba.

Aunque la literatura de Ingeniería de Software propone una y otra vez este tipo de procesos, en nuestra opinión las técnicas y herramientas concretas disponibles actualmente aun están muy lejos como para hacer del testing una actividad rentable para la mayoría de los equipos de desarrollo. Más aun, estamos convencidos de que la llave para habilitar estos procesos radica en contar con la posibilidad de automatizar el testing de unidad de alguna forma tal que los testers puedan comenzar a calcular los casos de prueba desde las fases iniciales del proyecto. Analizaremos una propuesta en este sentido en capítulos posteriores.

4. Testing de distintos aspectos de un software

Hasta aquí hemos tratado de manera implícita qué es lo que se verifica en un software. Por otro lado, hemos definido que un programa es correcto si verifica su especificación. Pero, ¿qué describe la especificación de un software? En Ingeniería de Software 1 al hablar de especificación hacíamos referencia únicamente a la función del software, es decir a las operaciones que el sistema debe realizar, el orden en que debe realizarlas, etc. Sin embargo, la especificación de un software, en términos más generales, incluye—o debería incluir—otros aspectos más allá de la funcionalidad—i.e. los requerimientos no funcionales. Por ejemplo, suelen especificarse requisitos de seguridad, desempeño, tolerancia a fallas, usabilidad, etc. Por lo tanto, al decir que un programa es correcto si verifica su especificación en realidad estamos incluyendo todos esos aspectos además de la funcionalidad. En consecuencia al testear un programa deberíamos seleccionar casos de prueba con el objetivo de testear todos esos aspectos y no solo la funcionalidad. Es decir no solo deberíamos intentar responder preguntas tales como "¿El sistema permite cargar una transacción solo en las condiciones especificadas?", sino también otras como "¿El sistema permite cargar hasta 10.000 transacciones por minuto?", "¿El sistema permite que solo los usuarios autorizados carguen transacciones?", etc.

Claramente todo esto es posible si el equipo de desarrollo cuenta con las especificaciones correspondientes. El testing de algunos de estos aspectos no funcionales se ha especializado en alguna medida dando lugar a áreas más específicas con sus propias metodologías, técnicas y herramientas. El caso más reconocido es, tal vez, el testing de seguridad el cual intenta encontrar errores que permitan a usuarios no autorizados utilizar o modificar datos o programas. El testing de este aspecto se ha especializado tanto que los testers de seguridad son profesionales con una preparación muy específica para realizar esta tarea.

A pesar de todas estas consideraciones en este capítulo solo abordaremos el testing de la funcionalidad de un software. Para mayores detalles sobre el testing de requisitos no funcionales pueden consultarse, entre otros, [6, capítulo 11] y [7, secciones 9.3, 9.6 y 9.9].

5. Las dos estrategias clásicas de testing

Tradicionalmente el testing de software se ha dividido en dos estrategias básicas que se supone son de aplicación universal.

- *Testing estructural o de caja blanca.* Testear un software siguiendo esta estrategia implica que se tiene en cuenta la estructura del código fuente del programa para seleccionar casos de prueba—es decir, el testing está guiado fundamentalmente por la existencia de sentencias tipo `if`, `case`, `while`, etc. En muchas ocasiones se pone tanto énfasis en la estructura del código que se ignora la especificación del programa [2], convirtiendo al testing en una tarea un tanto desprolija e inconsistente.

  Como los casos de prueba se calculan de acuerdo a la estructura del código, no es posible generarlos sino hasta que el programa ha sido terminado. Peor aun, debido a errores o cambios en las estructuras de datos o algoritmos—aun sin que haya cambiado la especificación—puede ser necesario volver a calcular todos los casos. Por estas razones preferimos la otra estrategia de testing, aunque estudiaremos el testing estructural con cierto detalle.

  Sin embargo es interesante remarcar la racionalidad detrás de esta estrategia: no se puede encontrar un error si no se ejecuta la línea de código donde se encuentra ese error. Aun así ejecutar una línea de código con algunos casos de prueba no garantiza encontrar un posible error; piense en una asignación de la forma `x = 1/y`. Por consiguiente es importante seleccionar casos de prueba que ejecuten al menos una vez todas las líneas del programa, lo cual se logra analizando la estructura del código.

  Se dice que el testing estructural prueba lo que el programa *hace* y no lo que se *supone que debe hacer*.

- *Testing basado en modelos (MBT)*[9] *o de caja negra.* Testear una pieza de software como una caja negra significa ejecutar el software sin considerar ningún detalle sobre cómo fue implementado. Esta estrategia se basa en seleccionar los casos de prueba analizando la especificación o modelo del programa, en lugar de su implementación [2].

  Algunos autores [5] consideran que el MBT es la automatización del testing de caja negra. Para lograr esta automatización el MBT requiere que los modelos sean formales dado que esta es la única forma que permite realizar múltiples análisis mecánicos sobre el texto de los modelos. Por el contrario, el testing de caja negra tradicional calcula los casos de prueba partiendo del documento de requerimientos.

  Como en el MBT los casos de prueba se calculan partiendo del modelo, es posible comenzar a "testear" casi desde el comienzo del proyecto—al menos mucho antes de que se haya terminado de programar la primera unidad. Por otra parte, pero por la misma razón, los casos de prueba calculados con técnicas de MBT son mucho más resistentes a los cambios en la implementación que aquellos calculados con técnicas de testing estructural. Más aun, el MBT es, efectivamente, automatizable en gran medida como demostraremos más adelante al utilizar una herramienta para tal fin. La gran desventaja del MBT radica en su misma definición: se requiere de un modelo formal del software, cosa que solo un porcentaje ínfimo de la industria es capaz de realizar. Debido al énfasis que le hemos dado a los métodos formales y debido a las ventajas que presenta el MBT es que estudiaremos con mucho detalle una técnica específica de esta estrategia de testing.

  Se dice que el testing basado en modelos prueba lo que el programa se *supone que debe hacer*, y no lo que el programa *hace*.

[9] 'model-based testing', en inglés.

Creemos que es muy importante remarcar que estas estrategias no son opuestas sino complementarias. En nuestra opinión el testing debería estar guiado fundamentalmente por técnicas de MBT pero complementadas con herramientas de análisis de cubrimiento de sentencias de forma tal que los casos generados mediante MBT cubran al menos todas las líneas de código.

En resumen, en los capítulos que siguen haremos hincapié casi exclusivamente en el *testing funcional* mediante técnicas de *MBT* para *testing de unidad*, aunque introduciremos algunos criterios de testing estructural (también para testing de unidad). Claramente esto es solo una fracción mínima de todo este tema pero en nuestra opinión es la base sobre la cual se puede construir el resto del marco teórico-práctico del testing.

Referencias

[1] I. Sommerville, *Software Engineering*, 9th ed. Harlow, England: Addison-Wesley, 2010.

[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of software engineering (2nd ed.)*. Prentice Hall, 2003.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, Apr. 2003.

[4] R. Dupuis, P. Bourque, A. Abran, J. W. Moore, and L. L. Tripp, "The SWEBOK Project: Guide to the software engineering body of knowledge," May 2001, stone Man Trial Version 1.00, http://www.swebok.org/ [01/12/2003].

[5] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[6] A. Page, K. Johnston, and B. Rollison, *How We Test Software at Microsoft*. Microsoft Press, 2008.

[7] S. L. Pfleeger, *Software Engineering: Theory and Practice*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.