Métodos Formales y Perspectiva Histórica de Prueba de *Software*

Pruebas de Sistemas

UNIVERSIDAD SIGLO 21

Métodos formales

Cuando hablamos de métodos formales, estamos hablando de aquellos métodos que aparecieron como puntos analíticos para poder verificar el desarrollo de *software* mediante la lógica de las matemáticas, que brindan una gran ventaja para poder mejorar la calidad de los sistemas y, por lo tanto, de lo que es la ingeniería de software.

Estos métodos se desarrollan y utilizan desde las fases iniciales del desarrollo de software y permiten que los errores surgidos en estas etapas tempranas puedan ser corregidos rápidamente y tener menos impacto en los costos y tiempos del proyecto.

¿Qué son los métodos formales?

El método formal es un procedimiento basado en matemáticas, utilizado para:

- especificación formal de sistemas, esto proporciona una descripción precisa de lo que debe hacer el programa de software;
- verificación formal, utilizando reglas precisas para demostrar que el software cumple la especificación formal detallada en el punto anterior;
- desarrollo formal, diseñando software de una forma segura que satisfaga las especificaciones formales.

Lo importante es que, si no hay una especificación formal del software, tanto la verificación como el desarrollo formal no podrán llevarse a cabo.

Estos métodos formales permiten plantear de un modo claro la especificación del sistema definiendo el comportamiento en términos de qué debe hacer el software y no de cómo lo debe hacer.

El propósito de estos métodos es sistematizar las fases de desarrollo para poder evitar que se pasen por alto errores críticos en los productos desarrollados y poder proporcionar un método estándar a lo largo del proyecto para que constituya un mecanismo preciso y no ambiguo.

Especificación formal de sistemas

La especificación formal es una descripción matemática fundamentada en el comportamiento del software. Utiliza tablas de estados o lógica matemática. Para las especificaciones, se reconocen los siguientes procesos:

> Lenguajes basados en modelos y estados. Permiten especificar el sistema mediante un concepto formal de estados y operaciones sobre estados. Los datos y relaciones/funciones se describen en detalle y sus propiedades se expresan en lógica de primer orden. La semántica de los lenguajes está basada en la teoría de conjuntos. (Fernández y Fernández et al., 2011, https://bit.ly/2u6o3Te)

Un ejemplo de este tipo de lenguaje es el Vienna Development Method (VDM), que define las especificaciones de software con un idioma propio del lenguaje que se llama *vienna definition language*. Se utiliza para declarar definiciones algebraicas de lenguajes de programación con una semántica operacional. El campo en donde se suele utilizar es aquel que requiere seguridad crítica (pasos de tren, centrales nucleares, etc.).

> Lenguajes basados en especificaciones algebraicas. Proponen una descripción de estructuras de datos estableciendo tipos y operaciones sobre esos tipos. Para cada tipo se define un conjunto de valores y operaciones sobre dichos valores. Las operaciones de un tipo se definen a través de un conjunto de axiomas o ecuaciones que especifican las restricciones que deben satisfacer las operaciones. (Fernández y Fernández et al., 2011, https://bit.ly/2u6o3Te).

Un ejemplo de este tipo de lenguaje es el TAD (tipo abstracto de dato). No requiere alta sofisticación matemática, es muy flexible, permitiendo modelar gran variedad de situaciones, y es general, es decir que se puede utilizar para cualquier tipo de sistemas.

> Lenguajes de especificación de comportamiento:
>
> - Métodos basados en álgebra de procesos: modelan la interacción entre procesos concurrentes. Esto ha potenciado su difusión en la especificación de

sistemas de comunicación (protocolos y servicios de telecomunicaciones) y de sistemas distribuidos y concurrentes.

- Métodos basados en Redes de Petri: una red de Petri es un formalismo basado en autómatas, es decir, un modelo formal basado en flujos de información. Permiten expresar eventos concurrentes.
- Métodos basados en lógica temporal: se usan para especificar sistemas concurrentes y reactivos. (Fernández y Fernández et al., 2011, https://bit.ly/2u6o3Te).

Un ejemplo de estos lenguajes es el Language of Temporal Ordering Specification (LOTOS). Es un lenguaje de especificación formal basado en la ordenación temporal de los eventos. Se usa, por ejemplo, para especificar protocolos en estándares ISO OSI.

Verificación formal

Las pruebas y verificaciones se deben dar entre:

- Las especificaciones y los requisitos: el objetivo es demostrar que el sistema, tal como está descripto en su especificación, cumple con los requerimientos establecidos para dicho sistema.
- El código fuente y la especificación: el objetivo es demostrar que el código fuente con el cual fue desarrollado el sistema cumple con la especificación formal escrita para dicho software.

Ventajas de los métodos formales

Especificación. La especificación formal proporciona mayor precisión en el desarrollo de software, y los métodos formales brindan las herramientas que pueden incrementar la garantía buscada. Desarrollar formalmente una especificación requiere conocimiento detallado y preciso del sistema, lo que ayuda a exponer errores y omisiones, y por lo que la mayor ventaja de los métodos formales se da en el desarrollo de la especificación (Clarke & Wing, 1996). En la formalización de la descripción del sistema se detectan ambigüedades y omisiones, y una especificación formal puede mejorar la comunicación entre ingenieros y clientes.

Verificación. Para asegurar la calidad de los sistemas es necesario probarlos, y para asegurar que se desarrollan pruebas rigurosas se requiere una precisa y completa descripción de sus funciones, incluso cuando en la especificación se utilicen métodos formales.

Aunque gran número de herramientas para automatizar pruebas se encuentran disponibles en el mercado, la mayoría automatiza sólo sus aspectos más simples: generan los datos de prueba, ingresan esos datos al sistema y reportan resultados. Definir la respuesta correcta del sistema, para un determinado conjunto de datos de entrada, es una tarea ardua, que la mayoría de herramientas no puede lograr cuando el comportamiento del mismo se especifica en lenguaje natural. Debido a que la respuesta esperada del sistema se puede determinar únicamente mediante la lectura de la especificación, los ingenieros esperan que a las pruebas automatizadas se les adicione este faltante y crítico componente (Dan & Aichernig, 2002). La gran ventaja de las herramientas, que generan pruebas con base en los métodos formales, es que la especificación formal describe matemáticamente el comportamiento del sistema, desde la que se puede generar la respuesta a un dato de entrada en particular, es decir, la herramienta puede generar casos de prueba completos.

Las técnicas de verificación formal dependen de especificaciones matemáticamente precisas y, desde un punto de vista costo-beneficio, generar pruebas desde la especificación puede ser uno de los usos más productivos de los métodos formales. Algunas mediciones empíricas demuestran que las pruebas, generadas con herramientas automatizadas, ofrecen una cobertura tan buena o mejor que la alcanzada por las manuales, por lo que los ingenieros pueden elegir entre producir más pruebas en el mismo tiempo, o reducir el número de horas necesarias para hacerlas.

Validación. Mientras que la verificación se puede realizar semiautomáticamente y las pruebas mecánicamente, la validación es un problema diferente. Una diferencia específica entre verificación y validación es que la primera responde a si “se está construyendo el producto correctamente”, y la segunda a si "se está construyendo el producto correcto" (Dasso & Funes, 2007). En otras palabras,

> la verificación es el conjunto de actividades que aseguran que el software implementa correctamente una función específica, y la validación es un conjunto de actividades diferentes que aseguran que el software construido corresponde con los requisitos del cliente (Amman & Offutt, 2008). Desde el conjunto de requisitos es posible verificar, formal o informalmente, si el sistema los implementa; sin embargo, la validación es necesariamente un proceso informal. Sólo el juicio humano puede determinar si el sistema que se especificó y desarrolló es el adecuado para el trabajo. A pesar de la necesidad de utilizar este juicio en el proceso de validación, los métodos formales tienen su lugar, especialmente en grandes y complejas aplicaciones, como en modelado y simulación. Una de sus aplicaciones más prometedoras es en el modelado de requisitos, ya que, al diseñarlos formalmente, el teorema provisto en la herramienta de prueba se puede utilizar para explorar sus propiedades, y a menudo detectar los conflictos entre ellos. Este método no sustituye al juicio humano, pero puede ayudar a determinar si se especificó el "sistema correcto", por lo que es más fácil determinar si las propiedades deseadas se mantienen. (Serna Montoya, 2010, https://bit.ly/2J7dTfF).

Conclusiones

Los métodos formales son “métodos rigurosos utilizados en el diseño y desarrollo de sistemas que utilizan matemática y lógica simbólica” (Cobo, 2014, https://bit.ly/2UstW90). Esta característica es la que brinda la formalidad del método.

Al utilizar estos métodos para la especificación y verificación del sistema, “incrementa la confianza del software que se está desarrollando” (Cobo, 2014, https://bit.ly/2UstW90).

Utilizan herramientas para probar matemáticamente que la ejecución formal del modelo desarrollado satisface los requerimientos formales especificados (Cobo, 2014).

Incrementan la detección de *bugs* tanto en el código como en la especificación, además de reducir los tiempos de todo el ciclo de vida del software.

Lo más importante es que, al utilizar métodos formales, incrementa la calidad de lo que se desarrolla.

La prueba de sistemas como una disciplina

Perspectiva histórica

Todo programa hace algo bien; pero tal vez sea lo que no queremos que haga. (Anónimo).

Es difícil brindar una definición clara de lo que son las pruebas de sistemas, ya que muchas prácticas se denominan de ese modo. Los distintos autores tienen criterios diferentes entre **lo que es hacer pruebas** y **lo que no es hacer pruebas**, además de los cambios y adaptaciones que tiene esta actividad, desde sus primeras referencias allá por 1950 hasta hoy, con las nuevas tecnologías y herramientas.

Actualmente, la palabra *test* o *testing* se emplea sin traducción para significar prueba, ya la tenemos incorporada en nuestro vocabulario.

Todos hemos realizado pruebas o test, por ejemplo, en la escuela, pruebas de esfuerzo para un estudio de corazón, pruebas para el ingreso universitario, pruebas para ingresar a un trabajo, etcétera. De este modo, entendemos a un test como una serie de preguntas o ejercicios para medir el conocimiento y las habilidades de un individuo. Si se extiende ese significado a la medición de los sistemas informáticos, el concepto no queda demasiado claro.

Si le pedimos a diferentes profesionales que indiquen lo que para ellos es hacer pruebas, obtendremos respuestas como las siguientes:

- Encontrar errores en programas.
- Determinar la aceptabilidad de usuarios.
- Adquirir confianza en el funcionamiento del sistema.
- Demostrar que no tiene errores.
- Evaluar las capacidades de un sistema.
- Verificar el funcionamiento de un programa.
- Asegurar que el sistema está listo para ser usado.

- Mostrar que el sistema funciona correctamente.
- Chequear el sistema contra las especificaciones.
- Asegurar la satisfacción del cliente.

Si se realiza este mismo ejercicio en cualquiera de las capacitaciones que se imparten de testing de *software* (pruebas de sistemas), se comprobará que, efectivamente, se dan muchísimas afirmaciones de lo que los profesionales, vinculados al proceso de desarrollo, creen o llevan adelante como **pruebas de sistemas**.

Todas estas proposiciones no son incorrectas, sino correctas aproximaciones, pero ¿cómo podemos definir a las pruebas formalmente?, ya que algunas de ellas denotan objetivos distintos. Por ejemplo, no es lo mismo decir que hacer pruebas es "demostrar que el sistema no tiene errores" que decir que es "mostrar que el sistema funciona correctamente". En ambas se involucran sentidos contrarios y objetivos distintos.

Vamos a revisar un poco de historia sobre el tema para brindar una definición y, a su vez, entender claramente el concepto.

Desde el momento en que comenzaron a escribirse los primeros programas, se puede decir que nacieron las pruebas. Un programa que se escribía para funcionar en una computadora tenía que ser probado y, si se encontraban errores, había que eliminarlos. Así surge la **primera visión**:

La prueba vista como una actividad siguiente a la programación, que implicaba no solo descubrir errores, sino también corregirlos y removerlos.

Recién a partir de 1960 es cuando comienzan las primeras distinciones entre lo que es hacer pruebas y depurar. Y también, durante esta década, es cuando la prueba de sistemas comienza a tomar más importancia.

Era ya evidente que los programas o sistemas contenían deficiencias y el costo e impacto de repararlos era significativo. Por esta razón, en la década de los 70 se comenzó a dedicar más tiempo y recursos a las pruebas.

Luego de la primera conferencia formal organizada en Estados Unidos, en junio de 1972, dedicada a las pruebas de sistemas, surgió la **segunda visión**:

La prueba de sistema abarca una serie de actividades asociadas con obtener la confianza de que un programa o sistema hace lo que se supone que tiene que hacer.

Debido a los resultados dispares obtenidos de las pruebas de sistemas, estas eran vistas como un elemento organizado, pero sin tecnología. A partir de entonces, se comienza a dedicar más tiempo a hablar de calidad, confiabilidad e ingeniería. Entonces, es cuando las pruebas de sistemas comienzan a emerger como una disciplina sistemática y metodológica.

Muchos textos, libros y conferencias comienzan a aportar a este desarrollo del tema, con la necesidad de muchos sectores de la industria de aplicar métodos más prácticos de aseguramiento de calidad.

Según Myers (1979), si se planteaban las pruebas con el objetivo de demostrar que el programa o el sistema no contenía errores, inconscientemente se iban a elegir datos para probar, con baja probabilidad de causar que el sistema falle. Surge así una **tercera visión** de las pruebas, según la cual el objetivo central es encontrar errores como parte de un proceso. Los datos seleccionados para probar son aquellos con alta probabilidad de tener errores.

"El testing es el proceso de ejecutar un programa o sistema con la intención de encontrar errores" (Myers, 1979, p. 5).

La anterior definición implicaba que solo se podía probar después de que el programa había sido codificado, con lo cual solo el código podía ser probado.

Si bien esta definición fue aceptada, en el ambiente de profesionales de ingeniería de software, se reconocía que existían otras maneras de probar un sistema, además de hacerlo a través de la ejecución.

Entonces, aparece una **cuarta visión**:

El testing como una actividad amplia y continua que acompaña el proceso de desarrollo.

Esto implica que cualquier actividad que es realizada con el objetivo de ayudar en la evaluación o medición de un atributo de software es considerada como una actividad de testing.

Una disciplina de testing efectiva ayuda a recolectar información que permita responder a preguntas como: ¿el software está listo para ser utilizado?, ¿cuáles son los riesgos?, ¿cuáles son sus limitaciones? ¿Funciona como se supone que tiene que funcionar? Es así como, en 1988, Bill Hetzel revisa su primera definición y la resuelve así: "El testing es cualquier actividad orientada a evaluar un atributo o capacidad de un programa o sistema y determinar si alcanza los resultados requeridos" (Hetzel, 1988, p. 6).

Al comenzar a hablar de capacidades, atributos o resultados requeridos, comenzó la relación con **calidad**, que significa cumplimiento de requisitos. Si los requerimientos están completos y un producto cumple esos requerimientos, entonces es un producto de calidad. Nace la **quinta visión** y define a las pruebas de sistemas como:

El proceso de verificar que el software se ajusta a los requerimientos y validar que las funciones se implementan correctamente.

Esta última atiende a tres puntos importantes:

1. Que su funcionamiento sea correcto implica que no hay errores que impidan operar el sistema.

2. Que el sistema cumpla con los requerimientos, es decir, que lo que se haya planteado esté desarrollado, que no falten funcionalidades o que no haya requerimientos incumplidos.

3. Que el hacer pruebas es visto como un proceso y no como una actividad al final de este.

De este modo, completamos el concepto de *prueba* e incorporamos también el concepto de *calidad*, a partir del cual se puede pensar en diversos factores relevantes que afectan la calidad del software.

Según Hetzel (1988), están, por un lado, los factores de calidad que tienen que ver con la funcionalidad y, por otro lado, los factores relacionados con cuán bien fue construido el software, incluyendo documentación, eficiencia, nivel de pruebas o facilidad de aprendizaje, entre otros, y por último, los factores que tienen que ver con la adaptabilidad, es decir, si se puede

adaptar a los cambios, a los diferentes entornos, y si se le puede incorporar una nueva funcionalidad cuando sea necesario.

Plantea tres dimensiones de la calidad del software (que los define como calidad exterior, interior y futura), con sus factores asociados:

- Funcionalidad definida como la calidad exterior:
  - Exactitud.
  - Usabilidad.
  - Confiabilidad.

- Ingeniería como calidad interior (tiene que ver con el proceso de construcción):
  - Capacidad de pruebas.
  - Eficiencia.
  - Documentación.

- Adaptabilidad como calidad futura:
  - Flexibilidad.
  - Reusabilidad.
  - Mantenibilidad.

La importancia de cada uno de estos factores varía según el tipo de sistema. Por ejemplo, un sistema de cuyo funcionamiento dependen vidas humanas atenderá de manera estricta a factores como la exactitud y la confiabilidad, mientras que un sistema comercial típico de oficina tendrá como factores relevantes la usabilidad y el mantenimiento. Entonces, una prueba efectiva debe estar orientada a proveer medidas de cada uno de los factores relevantes y hacer la calidad visible y tangible.

A partir de todo lo expuesto, podemos decir:

- Hacer pruebas no es demostrar que no hay errores.
- Hacer pruebas es detectar los errores y no buscar los motivos, ya que buscar los motivos es hacer *debugging*. Hacer pruebas no es *debbugear* un código, ya que podemos tener un sistema libre de errores, pero puede no servir para lo que se lo necesita.
- Hacer pruebas no implica simplemente verificar que las funciones requeridas se implementaron.
- Hacer pruebas es someter un software a ciertas condiciones que puedan demostrar si es o no válido a los requerimientos planteados.

- Considerar las pruebas es agregar valor no solo al producto, sino también al proceso de desarrollo, si se consideran los resultados generados.

En lo que se refiere a la historia, podemos resumirla a través de las visiones dadas:

1. **Primera visión**: inicialmente, hacer pruebas era hacer *debugging*.

   La prueba vista como una actividad siguiente a la programación, que implicaba descubrir errores, **corregirlos** y **removerlos** (Uninotas, 2017, https://bit.ly/2Uzfzjz).

2. **Segunda visión**: la prueba no es solo buscar errores o lo que no funciona bien, sino que se ve a la prueba de sistema como una serie de actividades asociadas a obtener la confianza de que un programa o sistema hace lo que se supone que tiene que hacer.

3. **Tercera visión**: las actividades de pruebas vistas como un proceso, en el que nunca se debe perder de vista el objetivo de encontrar errores. "El *testing* es el proceso de ejecutar un programa o sistema con la intención de encontrar errores" (Myers, 1979, p. 5).

4. **Cuarta visión**: en los 80 aparecen los **estándares** relacionados, que intentan aportar una metodología al proceso de pruebas y vincularlo directamente con el concepto de *calidad*, es decir, la prueba vista como cualquier actividad realizada con el objetivo de ayudar en la evaluación o medición de un atributo de software.

5. **Quinta visión**: la prueba vista como el proceso de verificar que el sistema desarrollado se ajuste a los requerimientos y de validar que las funciones se implementen correctamente. Incorpora tres puntos fundamentales: proceso, verificación cumplimiento de requerimientos y validación de funcionalidad.

6. En los 90 aparecen las **herramientas** para dar soporte al proceso y ayudar en la ejecución.

7. En el año 2000, comienza a considerarse como parte del **proceso de aseguramiento de calidad de *software***.

8. Si bien actualmente todavía suele ser un **proceso inmaduro**, esto está cambiando.

Respecto de las definiciones que se aportaron, con las distintas visiones de la prueba de sistemas, es importante destacar que se diferencian dos enfoques de pruebas: uno dinámico y otro estático.

El enfoque dinámico apunta a ejecutar una parte o todo el software para determinar si funciona según lo esperado, “mientras que el enfoque estático, se refiere a la evaluación del *software* sin ejecutarlo usando mecanismos automatizados (herramientas asistidas) y manuales, tales como controles de escritorio, inspecciones y revisiones” (Perez, 2007, https://bit.ly/2Pv2d4r).

De acuerdo con ambos enfoques, la prueba de sistemas se puede definir así:

Cualquier actividad realizada para evaluar la calidad del producto, identificar defectos y problemas y mejorar esa calidad, si se consideran los resultados obtenidos.

Esto implica que, por ejemplo, hacer inspecciones de código, controles de escritorio y revisiones de requerimientos es también hacer pruebas.

Según el enfoque dinámico, se define a las pruebas como:

La verificación dinámica del comportamiento de un programa contra el comportamiento esperado, por medio del uso de un conjunto finito de casos de prueba, seleccionados metodológicamente de un conjunto infinito de casos de ejecución. (Perez, 2007, https://bit.ly/2Pv2d4r)

Dinámica: implica que, para realizar las pruebas, hay que ejecutar el programa para los datos de entrada.

Comportamiento esperado: debe ser posible decidir cuando la salida observada de la ejecución del programa es aceptable o no, de otra forma el esfuerzo de la prueba es inútil. El comportamiento observado puede ser revisado contra las expectativas del usuario, contra una especificación o contra el comportamiento anticipado por requerimientos implícitos o expectativas razonables. (Perez, 2007, https://bit.ly/2Pv2d4r)

Podemos tomar otra definición del Institute of Electrical and Electronics Engineers (IEEE), en su Standard 610 (1990), que lo entiende como un proceso de operar el sistema o componente, bajo ciertas condiciones específicas, observar su comportamiento, registrar los resultados y realizar una evaluación de algún aspecto del sistema o componente.

Entonces:

Es el proceso de ejercitar el sistema para detectar errores y verificar que satisface las especificaciones de requerimientos funcionales y no funcionales.

Traemos esta definición porque menciona requerimientos funcionales y no funcionales, es decir, todo lo que tiene que ver con la función requerida y todo lo que no tiene que ver con la función, que normalmente no es requerido de manera explícita. Por ejemplo, si es fácil de usar, si las funciones se ejecutan en los tiempos necesarios y coherentes, si es adaptable, entre otros.

Referencias

Cobo, M. L. (2014). Métodos formales para Ingeniería de Software. Recuperado de http://cs.uns.edu.ar/~gis/mf14/downloads/Clases%20Teoria/Clase01(byn).pdf

Dasso, A. & Funes *A. (2007). Verification, validation and testing in software engineering.* Estados Unidos: Idea Group Publishing.

Drechsler, R. Ed. (2004). Advanced formal verification. Estados Unidos: Kluwer Academic Publishers

Fernández y Fernández, C., Ambrosio, M., Andrade, G., Cruz, G., José, M., Ortiz, R. y Sánchez, H. (2011). Métodos formales aplicados en la industria del software. *Temas de Ciencia y Tecnología*, *15*(43), 3-12. Recuperado de http://www.utm.mx/edi_anteriores/temas43/1ENSAYO_43_1-R.pdf

Hetzel, B. (1988). *An Introduction. The Complete Guide to Software Testing* (2nd ed). Massachusetts, US: QED Information Science.

Institute of Electrical and Electronics Engineers (IEEE). (1990). *Standard 610: Glossary of Software Engineering Terminology*. Recuperado de https://ieeexplore.ieee.org/document/159342/

Kaner, C., Bach, J., & Pretichord, B. (2001). *Lessons Learned in Software Testing*. New York, US: John Wiley & Sons.

Myers, G. (1979). *The Art of Software Testing.* New York, US: John Wiley & Sons.

Perez, B. (2007). *Testing exploratorio.* Recuperado de https://slideplayer.es/slide/1086289/

Serna Montoya, E. (2010). Métodos formales e ingeniería de software. *Revista Virtual Universidad Católica del Norte*, *30*, 158-184. Recuperado de https://bit.ly/2J7dTfF.

Uninotas (2017). *Prueba, implementaciòn y mantenimiento del sistema.* Recuperado de https://www.uninotas.net/prueba-implementacion-y-mantenimiento-del-sistema-3/

Wegner,P (1972) *The Vienna Definition Language.* Estados Unidos: ACM Comp Surveys

Jones, C. B (1990*) Systematic Software Development Using Vdm.* Estados Unidos: Prentice-hall International Series in Computer Science