Siete Mitos de los Métodos Formales*

Anthony Hall

Los métodos formales son polémicos. Sus defensores dicen que pueden revolucionar el desarrollo y sus detractores piensan que son extremadamente difíciles. Mientras tanto, para la mayor parte de las personas, los métodos formales son tan raros que les es difícil juzgar sus discutidos méritos. No existe mucha evidencia publicada que apoye una u otra postura, y mucho de lo que se comenta sobre los métodos formales se basa en opiniones, no en hechos. Por lo tanto, algunas creencias sobre los métodos formales han sido exageradas y casi han tomado el carácter de mitos. *Praxis* es una empresa de ingeniería de software donde hemos usado métodos formales en proyectos reales: escribimos especificaciones reales, no sólo ejercicios, y desarrollamos software real a partir de ellas. Como resultado de esta experiencia, muchos de nosotros somos grandes entusiastas de los métodos formales. Hemos hallado que ofrecen beneficios reales; también descubrimos que varias creencias sobre los métodos formales no son ciertas.

Este artículo da una mirada práctica a los métodos formales, presenta algunos de sus mitos –favorables y desfavorables– y explica lo que hemos encontrado tras ellos. Como ejemplo del uso de métodos formales, a lo largo de este artículo se usa el proyecto CASE, descrito en el anexo 1.

El proyecto CASE ciertamente no es el tipo de proyecto que la mayoría de las personas asociaría con el uso de métodos formales, y nosotros no hicimos un desarrollo completamente formal con demostraciones y verificación de programas. Sin embargo, hemos obtenido grandes beneficios al usar el lenguaje de especificación Z, que es tan solo uno de muchos lenguajes de especificación formal existentes.

Los siete mitos más generalizados sobre los métodos formales son variaciones de los siguientes:

1. *Los métodos formales garantizan que el software está perfecto.* El mito más importante es que los métodos formales serían todopoderosos, si nosotros humildes mortales pudiésemos aplicarlos. Este mito es pernicioso porque nos lleva a expectativas irreales y a la idea de que los métodos formales son de alguna forma todo-o-nada. La verdad es que no existe tal garantía, pero la utilidad de los métodos formales no depende de esta perfección absoluta.

2. *Los métodos formales se centran en demostrar correcci'on.* En los Estados Unidos, gran parte del trabajo desarrollado en métodos formales se ha concentrado en la verificación de programas. Esto ha hecho que los métodos formales parezcan muy difíciles y no muy relevantes para la vida real. Sin embargo, se puede lograr muchos beneficios aun sin hacer una sola demostración formal.

3. *Los métodos formales sonútiles sólo para sistemas críticos.* Esta creencia se basa en la percepción de la dificultad que implica la aplicación de métodos formales. La verdad es que los sistemas críticos requieren un uso más acucioso de métodos formales, pero cualquier sistema puede beneficiarse con el uso de algunas técnicas de especificación formal.

4. *Los métodos formales requieren matemáticos entrenados.* Los métodos formales se basan en notaciones matemáticas, y muchas personas creen que esto los hace difíciles para la práctica de los ingenieros de software. Este mito, a su vez, se basa en la percepción de que las matemáticas son intrínsicamente difíciles.

*IEEE Software, 7(5) 11 - 19, Septiembre 1990. Traducción de Cecilia Bastarrica, Mayo de 2001

En Praxis hemos hallado que las matemáticas de las especificaciones, al menos, son sencillas de aprender y utilizar.

5. *Los métodos formales aumentan el costo del desarrollo.* Se acostumbraba decir que a pesar que el costo que significaba usar métodos formales era muy alto, de todas formas era conveniente porque resultaba en menores costos de mantenimiento del software. Este argumento es difícilmente vendible a jefes de proyecto apremiados por los plazos, cuyo presupuesto es para desarrollo y no para mantenimiento. De hecho, tenemos cierta evidencia que el *desarrollo* puede resultar más barato si se usan métodos formales.

6. *Los métodos formales son incomprensibles para los usuarios.* Una especificación formal está llena de símbolos matemáticos que resultan incomprensibles para cualquiera que no esté familiarizado con la notación. De ahí que se suponga que son inútiles para clientes no matemáticos. Sin embargo, las matemáticas no son lo único en una especificación formal, sino que apoyan muchas otras formas de expresar la especificación que da al cliente una mejor comprensión del proyecto.

7. *Los métodos formales no se usan en grandes proyectos reales.* Los métodos formales se asocian comúnmente con departamentos académicos y organizaciones de investigación. Se piensa que sólo estas organizaciones tienen la capacidad necesaria para usar métodos formales y que éstos sólo son apropiados para las aplicaciones idealizadas que estos grupos desarrollan. Pero nuestra experiencia en el proyecto CASE, y la experiencia de otros usuarios de industria, están transformando este mito en historia.

Mito 1: Los métodos formales garantizan que el software es perfecto

El hecho es que los métodos formales son falibles. Debiera ser demasiado obvio para necesitar repetirse, pero nada puede alcanzar la perfección. Desafortunadamente, a veces los promotores de los métodos formales sostienen que estos ofrecen una garantía absoluta que no puede lograrse de otra forma. Si se toma esta posición, entonces cualquier problema con el software desarrollado formalmente es una refutación de la utilidad de los métodos formales. Los métodos formales han sido duramente criticados precisamente por estas bases absolutistas.

Es importante comprender las limitaciones intrínsecas de los métodos formales. Su falibilidad es su limitación más fundamental, y puede surgir de dos hechos: algunas cosas no pueden ser probadas y podemos cometer errores en las demostraciones de aquellas cosas que sí podemos probar.

Límites de las Demostraciones Una demostración es la prueba de que una especificación formal puede deducirse de otra. El mundo real no es un sistema formal. Una demostración, entonces, no muestra que en el mundo real las cosas sucederán como se espera. De modo que nunca se puede estar seguro que una especificación es "correcta", no importa cuanto se prueba acerca de ella.

Esto no debiera desanimarnos. Toda la ingeniería trata de hacer modelos del mundo real y usarlos para diseñar artefactos. Los modelos basados en las matemáticas son ideales porque pueden establecer las propiedades de los modelos mediante el razonamiento y porque pueden manipularse durante el diseño. El diseñador de una grúa por ejemplo, abstrae la grúa real a una estructura idealizada de componentes con propiedades conocidas de masa y capacidad de carga. Usa este modelo para predecir las propiedades de la grúa real, pero no hay forma de probar que la grúa real se comportará de acuerdo con las predicciones.

Pero en general la correspondencia entre los modelos matemáticos que se usan en ingeniería estructural y el mundo real son entendidos lo suficientemente bien como para confiar en ellos. Cuanto más madura es la disciplina de la ingeniería, más probable es que confiemos en los modelos que usa. Ha habido suficientes desastres de ingeniería como para convencer a cualquiera que la correspondencia no es perfecta, pero nadie sugeriría que los constructores de grúas abandonen las matemáticas.

En el software, las limitaciones de nuestras técnicas de modelamiento también son razonablemente comprendidas. Primeramente, los modelos cubren solamente algunos aspectos del comportamiento de los programas. Segundo, la correspondencia entre la descripción formal y el mundo real es limitada.

Existen buenos modelos matemáticos para el comportamiento de programas secuenciales. También existen modelos para comportamiento concurrente, pero no son tan fáciles de usar. Alguna personas dicen que no podemos modelar restricciones de tiempo formalmente; esto no es estrictamente cierto, pero es cierto que no sabemos bien como usar estos modelos para desarrollar software que satisfaga esas restricciones. Finalmente, aún no podemos modelar propiedades no funcionales tales como rendimiento, confiabilidad, mantenibilidad y disponibilidad.

La correspondencia entre nuestros modelos formales de los programas y el comportamiento real de los sistemas está limitado por tres factores: el comportamiento del lenguaje de programación, el sistema operativo, y el hardware subyacente. Para sistemas de seguridad crítica, estas limitaciones son cruciales y no podemos suponer que un programa es correcto tan solo porque ha sido probado.

Puede cometerse errores. Aun dentro de nuestro formalismo podemos cometer errores al realizar pruebas tanto como podemos cometer errores al escribir programas. Por cierto, muchas especificaciones formales publicadas contienen errores.

A pesar de estos problemas evidentes, los métodos formales *sí funcionan*. Existen dos razones para ello. Una es que hay algunas formas en las que los métodos formales ofrecen garantías cualitativamente diferentes y mejores que ningún otro método. La otra es que aunque los métodos formales todavía permiten cometer errores, son mucho más claros para dejar estos errores al descubierto.

Demostrar corrección. Existe una cita popular que dice que "las pruebas de programas pueden usarse para demostrar la presencia de un error pero nunca para mostrar su ausencia!" Esto parece implicar que alguna otra cosa –las demostraciones– pueden mostrar la ausencia de errores. Esto puede ser verdad de dos formas diferentes (a pesar de que en ambas la posibilidad de cometer errores en el proceso de razonamiento implica que las demostraciones no son absolutamente infalibles):

- Algunas propiedades pueden deducirse mediante razonamiento formal. Muchos requisitos son establecidos como verdades universales "El programa siempre registrará las acciones de los usuarios" y "El sistema nunca perderá un mensaje". Estas verdades en principio no pueden establecerse probando o simulando el sistema, sino mediante razonamiento sobre la especificación.
- Algunos pasos pueden ser demostradamente correctos. Por ejemplo, la relación entre un programa y su especificación es formal y puede demostrarse que es correcta. De modo que casi puede garantizarse que un programa corresponde a su especificación, a pesar de que esto no implica que el programa es perfecto. (La garantía es solamente "casi" debido a los límites del modelo matemático que captura el mundo real; aún si la garantía fuese absoluta, tampoco significaría que el programa está perfecto porque la especificación podría tener errores.)

Encontrar errores. A pesar de que eliminan solamente ciertos tipos de errores, los métodos formales hacen mucho más fácil encontrar todo tipo de errores. En una especificación informal es difícil decir qué es un error, porque no es claro qué se está diciendo. Al sentirse amenazadas, las personas tienden a defender sus especificaciones informales reinterpretándolas para eludir las críticas. Con una especificación formal, hemos encontrado que los errores son más fáciles de encontrar, y todos están más dispuestos a admitir que son errores.

En este sentido los métodos formales son un enfoque científico del desarrollo, dado que ofrecen especificaciones que pueden refutarse. (En desarrollo informal de software la especificación es generalmente refutada mediante

pruebas. En esta etapa de desarrollo, la especificación ya se ha formalizado traduciéndola a un lenguaje de programación, pero ya no es tan fácil de comprender por las personas.)

En el proyecto CASE y también en otros en que hemos usado métodos formales en Praxis, hemos encontrado que la habilidad de exponer los errores es uno de los beneficios esenciales de estos métodos. Sólo llevamos a cabo unas pocas demostraciones o pasos de desarrollo absolutamente formales, pero encontramos que las inspecciones de las especificaciones formales revelan más errores que en las especificaciones informales, y es más efectivo contrastar diseños o programas con sus especificaciones formales que con otros documentos de diseño. IBM ha reportado experiencias similares.

Mito 2: Los métodos formales se centran en demostrar corrección

El hecho es que los métodos formales se tratan de especificaciones.

Yo uso el término "métodos formales" para describir el uso de las matemáticas en el desarrollo de software. Las actividades principales que incluyo son:

- escribir la especificación formal,
- demostrar propiedades de la especificación,
- construir un programa mediante manipulación matemática de la especificación, y
- verfificar un programa mediante argumentos matemáticos.

Por lo tanto, la verficación de programas es sólo uno de los aspectos de los métodos formales. De muchas formas, es la parte más difícil. Para proyectos no críticos, la verificación de programas está lejos de ser el aspecto más importante de un desarrollo formal. Dado que el costo de remover errores crece dramáticamente a medida que el proyecto progresa, es más importante prestar atención a las fases iniciales.

Especificación de sistemas. Desde el punto de vista económico, por lo tanto, la parte más importante de un desarrollo formal es la *especificación del sistema*. Para muchos proyectos, esta es la única parte del desarrollo que es formal. En cualquier caso, una especificación formal de lo que un programa habrá de hacer es un requisito para verificar que el programa está correcto.

Una especificación formal es una definición precisa de lo que el software pretende hacer. Se puede hacer una especificación de cualquier parte del software, desde un módulo a un sistema completo. En el proyecto CASE, usamos Z para escribir la especificación formal del sistema completo. Dicha especificación de sistemas es la forma más práctica y valiosa de usar métodos formales.

Una especificación formal de un sistema es comparable en su alcance con un análisis de requerimientos convencional usando diagramas de flujo de datos o diagramas de entidad - relación. Se diferencian de las especificaciones convencionales en que sólo tienen relación con los requerimientos funcionales del sistema y no involucran ninguna decisión de diseño.

Para ilustrar la noción de una especificación formal, el anexo 2 muestra un ejemplo de una simplificación de parte de la especificación de CASE. Está escrita en Z. Una especificación Z es un modelo matemático del sistema a ser construido. Consiste de dos partes: la definición del estado del sistema y una colección de las operaciones del sistema.

Una especificación es abstracta de tres formas:

- usa tipos de datos tales como conjuntos y relaciones que pueden modelar las aplicaciones directamente más que tipos computacionales como arreglos. En el ejemplo, usé conjuntos para representar la colección de tareas y documentos en el sistema y una función para representar la relación entre ellos. Estas representaciones capturan la esencia de lo que se necesita más que las estructuras de implementación.
- especifica *qué* se va a hacer más que *cómo* se hará. La definición de *RemoverDocumento*, por ejemplo, dice simplemente que, después de la operación, el documento relevante ha sido removido. No dice nada acerca de cómo se realiza la remoción, ni cómo se encuentran y remueven las tareas relacionadas.
- sólo se especifica el nivel de detalle necesario; se puede dejar fuera las cosas que no son importantes. En el ejemplo, no dijimos cómo son realmente *TAREA* y *DOCUMENTO*. Esto también es un detalle de implementación que no tiene interés para el especificador.

Esta abstracción representa una separación de intereses apropiada entre lo que los usuarios quieren definir y lo que intentan dejar a la implementación. La separación de intereses es importante para controlar el proceso de desarrollo, independientemente del modelo de ciclo de vida elegido. Por ejemplo, en un desarrollo que usa prototipos para explorar los requerimientos del usuario, es importante separar el comportamiento esencial de la implementación del prototipo.

La incompletitud de la especificación puede solucionarse de dos formas. Primeramente, se debe registrar en otros documentos cosas tales como requerimientos no funcionales y otras cosas que quisiéramos decir al nivel de especificación pero no podemos debido a las limitaciones del modelo matemático usado. Segundo, durante los siguientes pasos de diseño e implementación se incluyen los detalles que se han dejado fuera.

Ocasionalmente estos pasos siguientes revelan problemas con la especificación que habían quedado ocultos debido a la abstracción. Por ejemplo, es posible escribir especificaciones que no pueden ser implementadas eficientemente. En este caso, se debe revisar la especificación durante el diseño.

La especificación es esencial para el proyecto en tres formas:

- el proceso mismo de construir la especificación es importante;
- la demostración de propiedades de la especificación es al menos útil como prueba de buena documentación;
- puede construirse una implementación correcta a partir de la especificación.

Beneficios. Hemos encontrado que escribir las especificaciones de CASE nos ayudó a clarificar los requerimientos, descubrir errores latentes y ambigüedades, y tomar decisiones acerca de la funcionalidad en la etapa apropiada.

Por ejemplo, comenzamos con requerimientos elaborados a partir de documentos con distinto status y transiciones complejas entre ellos. La formalización nos permitió simplificar el modelo a una pequeña serie de conceptos. Por ejemplo, modelamos separadamente el chequeo automático de los documentos de cuan lejos habían llegado en el proceso de aprobación. Esto hizo más fácil comprender y verificar con el usuario que las reglas que rigen los valores de status eran correctas. Esta clarificación de los requerimientos nos llevó a un sistema más pequeño y simple, y necesitamos menos trabajo durante las pruebas del sistema.

Es difícil plasmar una decisión al escribir una especificación formal, de modo que si hay errores o ambigüedades en su pensamiento, éstas serán reveladas sin piedad: usted encontrará que no puede escribir una especificación coherente o que, cuando le presenta la especificación a los usuarios, ellos rápidamente le dirán que está incorrecta. Es mejor ahora que cuando ya se haya incurrido en los costos de programación.

Muchas veces durante el desarrollo del proyecto CASE, descubrimos consecuencias inesperadas de la especificación. Por ejemplo, al principio escribimos una especificación que permitía que los documentos tuviesen versiones, pero no las tareas. Rápidamente descubrimos que no podíamos expresar este modelo formalmente. Para sobrellevar esto, introdujimos el concepto de versión de tareas, el cual representa una tarea en ejecución con un conjunto particular de versiones de documentos. Este concepto resutó representar un objeto del mundo real que era esencial para la forma en que la caja de herramientas de CASE sería usada, pero no nos habríamos dado cuenta de este uso claramente en una descripción informal del sistema.

Las especificaciones formales le permiten decir cualquier cosa que crea importante en la etapa de especificación. Al mismo tiempo, si realmente está preparado para dejar decisiones para una etapa posterior, también puede hacerse.

Nuestro ejemplo contiene una instancia típica de tal tipo de decisiones. Definimo en nuestra especificación precisamente lo que sucede cuando se obtiene la última salida de una tarea: la tarea es removida también. Es probable que en una especificación informal esto no hubiese sido tan claro y el codificador hubiese tenido que tomar la decisión. Pero esto es claramente un asunto de especificación dado que su efecto es visible para el usuario. Su omisión en la especificación formal, ya sea accidental o deliberada, sería obvia, porque habría una componente del estado cuyo valor estaría indefinido.

Especificaciones y demostraciones. Una vez que se tiene una especificación formal, se pueden demostrar cosas acerca de la especificación en sí, así como también demostrar que un programa las satisface. Estas otras propiedades pueden ser consistencia de la especificación o completitud de la definición de las operaciones. También pueden ser demostraciones de que la especificación (y por lo tanto el software desarrollado) tendrá ciertos requerimientos clave. Por seguridad, estos pueden ser cierto tipo de integridad u otros requerimientos. En todo caso, dado que los errores en esta etapa son más caros que los errores durante la implementación, la demostración de estas propiedades son también más importantes que las pruebas de implementación. Jim Woodcock ha mostrado razonamiento aplicado a especificaciones prácticas (el administrador de memoria CICS).

Implementación a partir de especificaciones formales. Cuando implementamos sistemas especifcados formalmente, no se hace escribiendo un programa y despu'es tratando de demostrar que satisface su especificación. Esto no es factible más que para pequeñísimos programas. En cambio, un programa correcto se construye en pequeños pasos. Cada paso toma la especificación y produce algo un poco más cercano al programa final. Cada pso es lo suficientemente chico como para que pueda verse claramente qué necesita demostrarse para probar que el paso es correcto, y si dudo de la correctitud puede realizarse la demostración. Este estilo de desarrollo se describe en los libros de texto como el Método Viena de Desarrollo de algoritmos correctos. Ha sido usado, por ejemplo, para implementar hardware a partir de especificaciones Z.

Cada paso de diseño en este tipo de desarrollo agrega algún detalle que fue omitido de la especificación formal o toma alguna decisión que se había pospuesto. Los implementadores deben:

- proporcionar estructuras de implementación eficiente para representar los conceptos de la aplicación,
- saber, conocer o desarrollar algoritmos para realizar las operaciones requeridas, y
- llenar los detalles que fueron deliberadamente dejados a su criterio y buen juicio.

En el proyecto CASE usamos formalidad solamente en la escritura de la especificación. No intentamos ningún tipo de demostración de programas. Los tipos de pasos de diseño que aplicamos fueron:

- decidir el lenguaje concreto de interfaz para las operaciones,

- decidir la estructura de datos concreta para representar estructuras abstractas de la especificación; por ejemplo, una clase de objetos para representar la función *outputTask*. El diseñador era libre para elegir cualquier representación que tuviese las propiedades requeridas.
- decidir ciertas operaciones de bajo nivel necesarias para implementar las operaciones de alto nivel. Por ejemplo, identificamos una componente llamada *kernel* que proporcionaba almacenamiento y funciones de distribución de bajo nivel. Especificamos esta componente formalmente y la implementamos a partir de su especificación Z.

Por supuesto que estos pasos de diseño requieren creatividad. La especificación no restringía a los diseñadores, pero tampoco les hacía el trabajo. Encontramos que tomar estas decisiones de diseño fue algo relativamente directo y que más aún, era fácil ver si un diseño propuesto satisfacía la especificación.

Una especificación es un tipo de contrato entre los especificadores y los implementadores, y si la especificación es formal, es fácil interpretar el contrato y decidir si se ha satisfecho.

Mito 3: Los métodos formales son útiles sólo para sistemas críticos

El hecho es que las especificaciones formales son útiles para cualquier sistema.

Probablemente la aplicación práctica más grande de métodos formales ha sido en sistemas no críticos. Nuestro proyecto CASE, por ejemplo, no era crítico en cuanto a seguridad. Los métodos formales deberían usarse siempre que el costo de fracaso es alto. Los sistemas cuyo costo de fracaso es alto incluyen aquellos que son:

- de alguna forma críticos,
- replicados muchas veces,
- fijos en el hardware, o
- dependientes de la calidad por motivos comerciales.

Casi cualquier software serio califica por alguno de estos motivos. Nuestro proyecto CASE, por ejemplo, tenía que ser un producto de alta calidad para satisfacer al cliente y a los usuarios.

La aplicación de métodos formales puede beneficiar muchas áreas, incluyendo lo apropiado que resulte el software para su propósito, su mantenibilidad, facilidad de construccón y mejor visibilidad.

La formalidad ofrece formas de asegurar que se construya el software apropiado. Usted puede discutir la especificación con el usuario y, en algunos casos, construir prototipos en base a la especificación para mostrar lo que se propone. Se puede usar razonamiento formal para mostrar algunas de las consecuencias de la especificacioón dándole algo concreto sobre lo cual discutir con el usuario.

Uno de los mayores problemas del mantenimiento de software es saber qué se supone que hará. Otro problema es saber qué se supone que hace cada parte, y por lo tanto qué debe preservarse cuando se realice el cambio. Las especificaciones formales son ideales para estos propósitos.

Nuestra experiencia muestra que es más fácil construir un sistema a partir de una especificación formal que usando otros métodos. Aún cuando no siguiésemos un desarrollo riguroso, encontramos que la codificación a partir de especificaciones formales es algo directo.

La aplicación de métodos formales permite también confiar más en su proceso de desarrollo porque en cada etapa es más claro qué se ha hecho y qué no se ha hecho. El monitoreo es más confiable y por lo tanto el desarrollo es menos riesgoso.

A partir de una especificación formal, el proceso de desarrollo puede ser muy riguroso si se hace en pequeños pasos, expresando y justificando formalmente cada paso. También puede ser menos riguroso si los pasos son más grandes y se justifican sólo informalmente. Usted elige el grado de rigor que mejor se ajuste a su aplicación. Si el sistema es crítico, deberá por cierto ser desarrollado formalmente.

Sin embargo, muchos beneficios de los métodos formales provienes de la estapa de la especificación. Por lo tanto, en sistemas no críticos, aún cuando ningún desarrollo sea formal, el solo hecho de escribir la especificación formal es una mejora importante sobre otros métodos informales.

Mito 4: Los métodos formales requieren matemáticos entrenados

El hecho es que las matemáticas de las especificaciones son fáciles.

Una vez que se reconoce que la práctica de los métodos formales se trata esencialmente de escribir especificaciones, las dificultades matemáticas resultan mucho menos relevantes. Se pueden desarrollar especificaciones con muy pocas matemáticas que cualquier ingeniero maneja.

Por ejemplo, en Z, las únicas ramas de las matemáticas que son necesarias para escribir especificaciones son la teoría de conjuntos y la lógica. Los elementos de ambas son sencillos y hoy en día son enseñadas a adolescentes.

Por cierto que antes que los ingenieros puedan usar métodos formales deben ser entrenados –en esto los métodos formales no son diferentes de otros métodos. Nuestra experiencia es que el entrenamiento no es difícil y personas con matemáticas de enseñanza media pueden escribir excelente especificaciones formales al ser entrenadas. Ciertamente cualquiera que pueda aprender un lenguaje de programación puede también aprender un lenguaje de especificación como Z.

La especificación de un problema es más corta y mucho más fácil de entender que su expresión en un lenguaje de programación. Considere la operación *RemoverDocumento* del anexo 2. La definición de esta operación en pseudocódigo sería mucho más larga y menos comprensible.

Las personas temen a los nuevos símbolos. Pero los símbolos matemáticos se usan para hacer las matemáticas más fáciles, no más difíciles. Las personas rápidamente se familiarizan con los nuevos símbolos. La disficultad de aprender lógica no está en los símbolos, así como la dificultad de aprender ruso no está en el alfabeto cirílico.

Dificultades. Esto no significa que todo es fácil al escribir especificaciones formales. Cuando se ha aprendido la notación, aún quedan dificultades. Algunas personas son más rápidas que otras, así como algunos son mejores programadores que otros.

La mayor dificultad radica en hacer las conexiones apropiadas entre el mundo real y el formalismo matemático. Puede ser difícil elegir las cosas apropiadas del mundo real para modelar -captar el nivel apropiado de abstracción. Algunos programadores ponen demasiados detalles en sus especificaciones y las vuelven muy complejas. También se puede cometer el error inverso: escribir especificaciones demasiado abstractas.

Sin embargo, estos problemas están en toda especificación, no son introducidos por la formalidad. Muchos programadores tienen dificultades en escribir una especificación usando una notación porque les es difícil apartarse de los detalles de los lenguajes de programación. Cuando se usan especificaciones formales, el estudio de buenos casos publicados y el consejo de personas experimentadas puede ayudar a evitar estos problemas.

Pautas de entrenamiento. Hemos encontrado que existen tres etapas de entrenamiento necesario:

- entrenamiento en matemáticas discretas, que requiere cubrir teoría de conjuntos elemental y lógica formal. Para aquellos con conocimientos de matemáticas pero no familiarizados con estos temas, un sólo día es suficiente para presentar estas ideas. Aún para los que tienen menos conocimientos previos, se requiere menos de una semana de entrenamiento. Existen muchos buenos libros de matemáticas discretas.
- entrenamiento en la notación formal particular que se use. Un curso de Z o VDM toma típicamente dos semanas, suponiendo que los participantes tienen los conocimientos necesarios de matemáticas. Existen textos de VDM y Z.
- ejercitar con proyectos reales. Después del entrenamiento, los estudiantes pueden usar los métodos formales, pero aún encontrarán dificultades. Para superarlas, recomendamos hacer talleres donde pueden atacarse los problemas con la ayuda de un tutor. También es esencial que todo proyecto que use métodos formales tenga acceso al menos a una persona con experiencia en el uso del método. Si fuese necesario, podría asegurar esto contratando un consultor durante las primeras etapas del proyecto: 10 días hombre usados sabiamente pueden ser suficientes.

Un nivel mucho más avanzado de matemáticas se requiere si se intenta ir más alla de las especificaciones formales y hacer un desarrollo completamente formal que incluya demostraciones de corrección. No es realista esperar que la mayor parte de los ingenieros de software estén capacitados para realizar demostraciones fácilmente. Tampoco es probable que ayude mucho la automatización. Las herramientas de demostración están aún en un estado muy primitivo –y, si acaso, existen dificultades fundamentales con las demostraciones automáticas.

Por lo tanto, personas competentes que pueden lidiar con la manipulación matemática necesaria son aquellos que deben llevar a cabo los proyectos de seguridad crítica. Por supuesto que también esto es cierto para construir un puente.

Mito 5: Los métodos formales aumentan los costos de desarrollo

El hecho es que escribir especificaciones formales disminuye el costo de desarrollo.

Un desarrollo completamente formal incluye demostraciones de cada paso de desarrollo que son muy caras - probablemente no factibles de modo que sólo valen la pena para las aplicaciones más críticas. Pero dado que muchos beneficios provienen de la escritura de las especificaciones formales en s'i, es importante saber si ésto es muy caro.

Menor costo de desarrollo. Es notablemente difícil comparar los costos de desarrollar software con distintos métodos. No existen cifras para los costos de desarrollo del mismo software usando un método formal bien establecido y un método informal comparable. Sin embargo, ya se está acumulando experiencia acerca del costo de los proyectos que usan especificaciones formales. Ninguno de estas evidencias apoyan la idea de que los costos de desarrollo son mayores si se usan especificaciones formales; si acaso, algunas sugieren que los costos son menores.

Nuestra propia experiencia en el proyecto CASE mostró que la productividad (medida en líneas de código por día) medida desde el comienzo de la especificación hasta la aceptación final fue mucho mayor que nuestras cifras estimadas normales. Debido a que implementamos el proyecto CASE en un lenguaje productivo (Objective C), la tasa de productividad en términos de funcionalidad implementada por día probablemente sería aún mayor.

La empresa Rolls-Royce & Associates ha reportado que en proyectos de seguridad crítica donde se usaron especificaciones formales y pruebas planeadas, se obtuvieron mejores cifras de productividad que cuando no se usaron. (Al principio, la productividad fue menor, pero esto se atribuye al aprendizaje de usar varias herramientas no muy amigables y no estaba relacionado con el método formal en sí.) El costo de aprender a usar un método formal no era un problema importante, a pesar que IBM ha destacado este aprendizaje como un importante costo que se realiza una sola vez. Rolls Royce reportó que el 7 % del tiempo usado en especificación permitió evitar grandes costos al final del proyecto.

Cambios en el ciclo de vida. A pesar de que usar especificaciones formales en un proyecto no lo hace más costoso, sí cambia la forma del proyecto. Se usa más tiempo en la fase de especificación –en el proyecto CASE, se usó cerca del 30 % del esfuerzo total antes de comenzar la implementación. Por qué? Porque se hace más trabajo en esta etapa que lo que es habitual. Pero las etapas de implementación, integración y pruebas son más cortas.

Esta etapa de especificación más larga causa problemas: puede ser difícil administrar el proceso de especificación porque es más difícil ver si se ha progresado. Especialmente al principio puede ser difícil creer que se tiene algún progreso en absoluto dado que todo tipo de ideas son lanzadas –lo cual es como debe ser. Nuestra experiencia sobre la forma de crecimiento de la especificación se muestra en la figura 1.

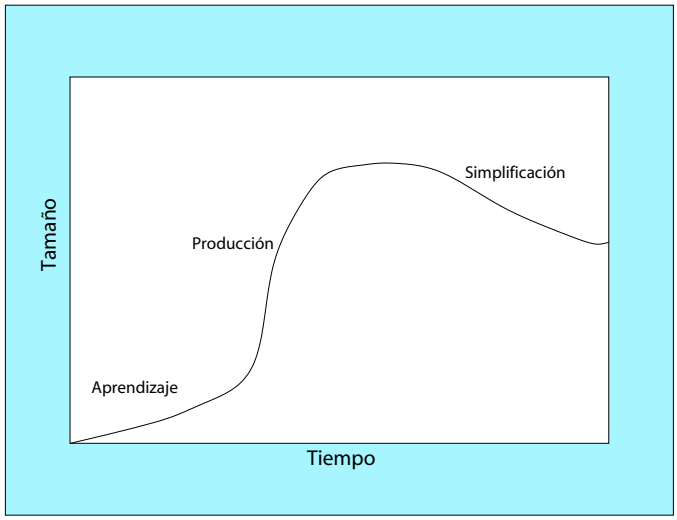

Figura 1: Historia de vida de una especificación.

Al principio parece que pasan muy pocas cosas. Pero después de un tiempo, la gente comienza a entender el problema y se progresa rápidamente. Luego el crecimiento se hace más lento y, si todo marcha adecuadamente, la especificación comienza a decrecer. Esto es cuando el problema comienza a ser comprendido realmente y donde las regularidades y similitudes se reconocen, lo cual hace que la especificación sea más compacta y mejor. Este proceso de pulido puede continuar indefinidamente y un buen administrador de proyecto debe saber decidir cuando detenerlo. Ciertamente no debe detenerse mientras la especificación aún está creciendo –en este momento el problema no ha sido suficientemente analizado.

Es importante registrar las especificaciones plausibles que se intentaron y rechazaron, así como también las razones de tal rechazo, no sólo la especificación final. Estos registros ayudarán a guiar futuros proyectos, previniendo la repetición de trabajo infructuoso, y también como guía a los mantenedores del sistema.

Es imprescindible reconocer también que las especificaciones nunca son perfectas. Cuando se llega a la etapa de implementación, hallará siempre deficiencias en la especificación. Cuando esto sucede, se debe modificar la especi-

ficación –aplicando control de cambios por supuesto. Siempre existe la tentación de corregir la implementación dejando la especificación como está –esto lleva rápidamente a la divergencia entre la especificación y el software real e implica que la especificación será inútil para el mantenimiento. Ambos deben mantenerse en sintonía. Si se hace esto, la especificación sigue siendo un documento valioso a lo largo del ciclo de vida del software. Claramente, hay un costo en hacer esto, pero no es muy grande: en el proyecto CASE, fue menos del 5 % del esfuerzo de la etapa de implementación.

Mito 6: Los métodos formales son incomprensibles para los usuarios

El hecho es que las especificaciones formales ayudan a los usuarios a entender lo que están obteniendo.

Cómo? La especificación capta lo que los usuarios quieren antes de construirlo. Pero para darse cuenta de sus beneficios, se debe hacer la especificación comprensible para los usuarios. Existen tres formas de hacer esto:

- reescribir la especificación en lenguaje natural,
- mostrar las consecuencias de la especificación,
- animar la especificación.

La primera de estas formas es siempre esencial. Una especificación matemática siempre debe ser acompañada de una descripción en lenguaje natural que explique el significado de la especificación en términos del mundo real y por qué la especificación dice lo que dice.

Debe asignarse tiempo y recursos para el esfuerzo de escribir el texto acompañante. Este esfuerzo vale la pena, dado que la experiencia ha mostrado que la documentación producida a partir de una especificación formal es más comprnsible, más exacta, más corta, y más útil que las especificaciones informales.

En una especificación formal bien producida, se pueden quitar completamente las partes matemáticas –el resultado es un documento en lenguaje natural que es una especificación mucho mejor del sistema que una especificación formal convencional. También puede usarse especificaciones formales con notación de diagramas –nada previene el uso de ninguna notación que facilite la descripción del sistema.

Una forma en que las especificaciones formales son más útiles que ningún otro método es que pueden permitir mostrar mediante razonamiento formal al usuario que el sistema satisface ciertos requerimientos. Sólo puede hacerse esto si los requerimientos se expresan formalmente, pero muchas propiedades tales como la seguridad pueden expresarse sólo en forma parcialmente formal. Aún cuando no existan requerimientos expresados formalmente, se pueden deducir ciertas consecuencias de la especificación y presentárselas al usuario. En el proyecto CASE, dedujimos (a pesar de que no lo demostramos formalmente) propiedades tales como “ninguna versión almacenada en la máquina del proyecto cambia”.

Las especificaciones formales son algo concebido como antitéticas a técnicas como animación y prototipación. De hecho los enfoques son complementarios, y ambos persiguen el objetivo de establecer los requerimientos del usuario con mayor confiabilidad. Una forma de reunirlos es construir prototipos para explorar los requerimientos y luego registrar los resultados en una especificación formal como base del desarrollo posterior. Algunas veces se puede usar prototipos para definir áreas que no están bien expresadas en la especificación formal. En el proyecto CASE, usamos prototipos para explorar detalles de la interfaz con el usuario y especificaciones formales para la funcionalidad del sistema. La notación formal puede animarse dando la capacidad de tener prototipos inmediatamente. Sin embargo, los lenguajes de especificación más poderosos no pueden ejecutarse de esta forma, y se requiere de otro paso independiente como la implementación de Prolog, para animar la especificación.

Mito 7: Los métodos formales no se usan en grandes proyectos reales

El hecho es que los métodos formales se usan solamente en proyectos industriales.

Muchas organizaciones, no solamente Praxis, están usando métodos formales en proyectos industriales. Muchas personas conocen aplicaciones en el área de seguridad, pero el alcance de los métodos formales es mayor. Los siguientes son sólo algunos de los tipos de proyectos que están usando métodos formales:

- Procesamiento de transacciones. El sistema CICS de IBM, es un sistema grande de procesamiento de transacciones con más de 20 años de antigüedad. Contiene más de medio millón de líneas de código. IBM está usando Z para especificar las interfaces clave de CICS para mejorar su mantenibilidad. Hasta ahora, las especificaciones Z se han escrito para más de 100.000 líneas de código nuevo o modificado.
- Hardware. El uso de métodos formales no está confinado al software. Existen al menos tres ejemplos de notación Z usada para especificar hardware. Una de ellas es el Multiprocesador Seguro de Información de de la arquitectura de computador segura de spe Environment. El código de orden de SMITE ha sido especificado en Z por la compañía inglesa Odessey. La unidad de punto flotante del Transputer ha sido especificada en Z, incidentalmente revelando varios errores en otras implementaciones de punto flotante. Tektronix ha usado Z para especificar la funcionalidad de familias de osciloscopios.
- Compitadores. El DAnish Datamatik Center ha desarrollado por muchos años compiladores industriales usando métodos formales.
- Herramientas de software. Nuestro proyecto CASE es sólo una, a pesar de que es el ejemplo más completo del uso de métodos formales para el desarrollo de herramientas. Otros ejemplos son la interfaz del Portable Common Tools Environment, un estándar europeo de ingeniería de software, y la verificación de ambientes de ingeniería de software para bases de datos.
- Control de reactores. Rolls Royce and Associates usaron una combinación de inglés y especificaciones formales para especificar un software de control de un reactor nuclear. Usaba animación para explorar la especificación con el ingeniero responsable.

Claramente, estos proyectos representan una pequeñísima fracción de todo el desarrollo de software. Sin embargo, son aplicaciones reales de escala industrial, y ellos reportan beneficios positivos del uso de métodos formales.

Nuestra propia experiencia en el proyecto CASE ha sido que los métodos formales pueden ser muy efectivos. Pero son solamente una parte de un proyecto: el proyecto CASE usó especificaciones formales en el marco de aseguramiento de calidad y control y administración de proyectos, con otras buenas técnicas de diseño, implementación y pruebas.

Los métodos formales no ofrecen ninguna garantía mágica: nuestro proyecto CASE era algo normal con sus problemas esperables. Pero el equipo del proyecto cree que la formalidad de la especificación fue de beneficio sustancial a lo largo de todo el proyecto.

Como resultado de nuestra experiencia, creemos que los métodos formales deben ser mejor entendidos por los desarrolladores en general. Son herramientas poderosa, pese a no ser una panacea. Las razones de su efectividad no son necesariamente las razones por las que fueron desarrollados. Tampoco la dificultad en su uso está en la notación matemática sofisticada que usan.

En lugar de perpetuar los siete mitos, ofrezco siete hechos para reemplazarlos:

1. Los métodos formales son muy útiles para encontrar errores tempranamente y pueden casi eliminar cierto tipo de errores.

2. Trabajan en términos generales haciendo pensar muy detalladamente acerca del sistema que se pretende construir.
3. Son útiles para la mayor parte de las aplicaciones.
4. Se basan en especificaciones matemáticas que son mucho más sencillas de comprender que los programas.
5. Pueden hacer disminuir el costo de desarrollo.
6. Pueden ayudar a los clientes a comprender lo que están comprando.
7. Han sido usados con éxito en proyectos industriales.

1. Anexo: El Proyecto CASE

El proyecto CASE donde aplicamos especificaciones formales es una herramienta de ingeniería de software para apoyar equipos de trabajo que usen SSADM, un método de análisis y diseño estructurado. Cada miembro del equipo tiene una estación de trabajo, y la estaciones están conectadas en red a una máquina central del proyecto. La infraestructura del proyecto CASE proporciona:

- un sistema de administración de múltiples usuarios del proyecto distribuidos y un sistema de control de la configuración para controlar toda la información y las tareas de desarrollo, y
- un conjunto de clases básicas (tales como diagramas, tablas y matrices) a partir de las cuales se desarrollan las herramientas para análisis estructurado mediante especialización.

La infraestructura se implementa sobre Sun Unix. Se codifica en Objective C.

La especificación se documenta en alrededor de 340 páginas escritas en Z con comentarios en inglés. Contiene alrededor de 550 esquemas que definen 280 operaciones.

El desarrollo de las especificaciones se realizó mediante:

- escribir una especificación concreta de las interfaces en Objective C,
- escribir, para alguna partes del sistema, documentos de diseño informal,
- codificar otras partes directamente a partir de las especificaciones Z,
- escribir algunas especificaciones Z para módulos de bajo nivel, y
- codificar a partir de los diseños informales o las especificaciones de bajo nivel.

Usamos los estándares propios de nuestra compañía para la planificación, integración, pruebas y control de la configuración del proyecto.

Codificamos alrededor de 58.000 líneas de Objective C, de las cuales alrededor de 37.000 fueros parte del software final.

El proyecto duró 90 semanas y fueron usados alrededor de 450 semanas-hombre de esfuerzo, de las cuales alrededor de dos se dedicaron a la especificación del sistema.

2. Anexo: Un ejemplo de especificación formal

El sistema del proyecto CASE contiene una colección de documentos y una colección de tareas. Cada documento es producido por una tarea; las tareas pueden producir más de un documento; todas las tareas producen al menos un documento.

Para describir esto en Z hemos construido un modelo matemático. No dijimos cuáles tareas y documentos son, de modo que nos permitiese representarlos con los nombres TAREA y DOC en esta etapa. En la notación Z, el texto de la primera parte del escquema es la declaración, el cual describe las componentes del modelo; el texto en la segunda parte del esquema es el predicado que describe las propiedades del modelo. Los esquemas se dividen con una línea horizontal.

Definición de tareas y documentos. Esta parte del modelo se llama *TareasYDocumentos*. La especificación es la siguiente.

TareasYDocumentos

$$documentos : \mathbb{P}\, DOC$$
$$tareas : \mathbb{P}\, TAREA$$
$$tareaOutput : DOC \pfun TAREA$$

$$\mathrm{dom}\, tareaOutput = documentos$$
$$\mathrm{ran}\, tareaOutput = tareas$$

En Z, el símbolo para un conjunto es $\mathbb{P}$, que se lee como "conjunto de". Las dos primeras líneas de nuestro modelo definen las componentes documentos, que es un conjunto de DOCs, y las componentes tareas, que es un conjunto de TAREAs. Esto expresa el hecho de que "el sistema contiene una colección de documento y una colección de tareas".

Luego, se debe decir que "cada documento es producido por una tarea". Hicimos esto en dos partes. Primero establecimos una asociación entre los documentos y las tareas que los producen, que llamamos *tareaOutput*. Esta asociación se describe como una función para lo cual Z usa el símbolo $\pfun$, que indica que un documento puede ser el producto de una sola tarea.

Entonces se debe decir que cada documento se produce de esta forma, de modo que decimos que la acción asocia todos los documentos conocidos con las tareas: esto se establece en la línea "$\mathrm{dom}\, tareaOutput = documentos$", porque la expresión "$\mathrm{dom}\, tareaOutput$" significa "todos los documentos que se asocian con tareas mediante la función *tareaOutput*.

Similarmente, la expresión Z "$\mathrm{ran}\, tareaOutput$" significa "todas las tarea que se asocian con documentos en la función *tareaOutput*". Para expresar que los requisitos de que todas las tareas produzcan al menos un documento decimos que "$\mathrm{ran}\, tareaOutput = tareas$".