Software Aging

Invited Plenary Talk

David Lorge Parnas

Communications Research Laboratory
Department of Electrical and Computer Engineering
McMaster University, Hamilton, Ontario, Canada L8S 4K1

ABSTRACT

Los programas, al igual que las personas, envejecen. No podemos evitar el envejecimiento, pero podemos entender sus causas, tomar medidas para limitar sus efectos, revertir temporalmente algunos de los daños que ha causado y prepararnos para el día en que el software ya no sea viable. Una señal de que la profesión de Ingeniería de Software ha madurado será que perdamos nuestra preocupación por la primera versión y nos centremos en la salud a largo plazo de nuestros productos. Los investigadores y profesionales deben cambiar su percepción de los problemas del desarrollo de software. Solo entonces la Ingeniería de Software merecerá ser llamada Ingeniería.

1. ¡Qué tontería!

Puedo imaginar fácilmente la reacción de algunos científicos de la computación ante el título de este documento. "El software es un producto matemático; las matemáticas no se degradan con el tiempo. Si un teorema era correcto hace 200 años, seguirá siendo correcto mañana. Si un programa es correcto hoy, lo será dentro de 100 años. Si está mal dentro de 100 años, debe haber estado mal cuando se escribió. No tiene sentido hablar del envejecimiento del software".

Como muchas afirmaciones de este tipo, la cita imaginada es verdadera pero no realmente relevante. Los productos de software muestran un fenómeno que se asemeja estrechamente al envejecimiento humano. El software antiguo ha empezado a perjudicar a sus antiguos orgullosos propietarios; muchos productos ahora son vistos como un legado molesto del pasado. Se está destinando un esfuerzo cada vez mayor al soporte de estos productos más antiguos. Al igual que el envejecimiento humano, el envejecimiento del software es Inevitable, pero al igual que el envejecimiento humano, hay cosas que podemos hacer para frenar el proceso y, a veces, incluso revertir sus efectos. El envejecimiento del software no es un fenómeno nuevo, pero está adquiriendo importancia debido a la creciente relevancia económica del software y al hecho de que, cada vez más, el software es una parte importante del "capital" de muchas empresas de alta tecnología. Muchos productos de software antiguos se han vuelto piezas esenciales en la maquinaria de nuestra sociedad. El envejecimiento de estos productos está obstaculizando el desarrollo continuo de los sistemas que los incluyen.

Los autores y propietarios de nuevos productos de software a menudo miran con desprecio al software envejecido. Creen que, si el producto hubiera sido diseñado utilizando las técnicas actuales, no estaría causando problemas. Tales comentarios me recuerdan a un joven corredor burlándose de un hombre de 86 años (que, desconocido para el corredor, fue un nadador campeón hasta los 50) y diciendo que debería haber hecho más ejercicio en su juventud.

Así como todos envejeceremos (si tenemos suerte), el envejecimiento del software puede y ocurrirá en todos los productos exitosos. Debemos reconocer que sucederá con nuestros productos y prepararnos para ello. Cuando llegue la vejez, debemos estar preparados para enfrentarla.

El propósito de este documento es explicar cómo un producto abstracto y matemático puede envejecer y luego revisar algunos enfoques para lidiar con ello.

1. Las causas del envejecimiento del software

Existen dos tipos bastante distintos de envejecimiento del software. El primero se debe al fracaso de los propietarios del producto al modificarlo para satisfacer las cambiantes necesidades; el segundo es el resultado de los cambios que se realizan. Este "golpe uno-dos" puede llevar a un rápido declive en el valor de un producto de software.

* 1. Falta de movimiento

En las últimas tres décadas, nuestras expectativas sobre los productos de software han cambiado enormemente. Recuerdo los días en que un programador "parcheaba" un programa almacenado en cinta de papel usando pegamento y papel. Estábamos dispuestos a presentar grandes montones de tarjetas y esperar horas o días para que el trabajo se compilara y ejecutara. Cuando apareció la programación interactiva, estábamos dispuestos a usar lenguajes de comandos crípticos. Hoy en día, todos dan por sentado el acceso en línea, la respuesta "instantánea" y las interfaces basadas en menús. Esperamos capacidades de comunicación, almacenamiento masivo en línea, etc. El primer producto de software que construí (en 1960) haría su trabajo perfectamente hoy (si pudiera encontrar una computadora Bendix), pero nadie lo usaría. Ese software ha envejecido incluso aunque nadie lo haya tocado. Aunque los usuarios en la década de 1960 estaban entusiasmados con el producto, los usuarios de hoy esperan más. Mi antiguo software, en el mejor de los casos, podría ser el núcleo de un sistema más conveniente en el mercado actual. A menos que el software se actualice con frecuencia, los usuarios se sentirán insatisfechos y cambiarán a un nuevo producto tan pronto como los beneficios superen los costos de la capacitación y la conversión. Se referirán a ese software como antiguo y obsoleto.

* 1. Cirugía ignorante

Aunque es esencial actualizar el software para prevenir el envejecimiento, cambiar el software puede causar una forma diferente de envejecimiento. El diseñador de un software generalmente tenía un concepto simple en mente al escribir el programa. Si el programa es grande, entender ese concepto permite encontrar las secciones del programa que deben modificarse cuando se necesita una actualización o corrección. Comprender ese concepto también implica comprender las interfaces utilizadas dentro del sistema y entre el sistema y su entorno. Los cambios realizados por personas que no comprenden el concepto de diseño original casi siempre provocan que la estructura del programa se degrade. En esas circunstancias, los cambios serán inconsistentes con el concepto original; de hecho, invalidarán el concepto original. A veces, el daño es pequeño, pero a menudo es bastante severo. Después de esos cambios, se deben conocer tanto las reglas de diseño originales como las nuevas excepciones introducidas a las reglas para entender el producto. Después de muchos cambios de este tipo, los diseñadores originales ya no comprenden el producto. Aquellos que hicieron los cambios, nunca lo hicieron. En otras palabras, nadie comprende el producto modificado. El software que ha sido modificado repetidamente (mantenido) de esta manera se vuelve muy costoso de actualizar. Los cambios llevan más tiempo y es más probable que introduzcan nuevos "errores". El envejecimiento inducido por el cambio a menudo se agrava por el hecho de que los encargados sienten que no tienen tiempo para actualizar la documentación. La documentación se vuelve cada vez más inexacta, dificultando aún más los cambios futuros.

1. Fallo renal

Un problema que a menudo se confunde, pero es distinto del envejecimiento del software, es la desaceleración del sistema causada por la falta de liberación de memoria asignada. Los archivos pueden crecer y requerir poda. A veces, una rutina de asignación de memoria puede no liberar todo el espacio que se ha asignado. Lentamente, se disminuyen el espacio de intercambio y el espacio de archivo y el rendimiento se degrada. Este problema a menudo es una falla de diseño congénita y puede ocurrir a cualquier edad; pero también puede ser el resultado de una cirugía ignorante o agravarse por patrones de uso cambiantes. Sin embargo, se cura más fácilmente que el "envejecimiento" que es el tema de este documento. Un proceso tipo diálisis puede intervenir y limpiar el sistema de archivos y la memoria, rutinas mejoradas pueden hacer que la limpieza ocurra rápidamente y el software puede considerarse completamente "curado".

1. Los costos del envejecimiento del software

Los síntomas del envejecimiento del software reflejan los del envejecimiento humano: (1) los propietarios de software envejecido encuentran cada vez más difícil mantenerse al día con el mercado y pierden clientes frente a productos más nuevos, (2) el software envejecido a menudo se degrada en su rendimiento espacio/temporal como resultado de una estructura que se deteriora gradualmente, (3) el software envejecido a menudo se vuelve "defectuoso" debido a errores introducidos cuando se realizan cambios. Cada uno de estos resultados tiene costos reales para el propietario.

* 1. Incapacidad para mantenerse al día

A medida que el software envejece, crece en tamaño. Este "aumento de peso" es el resultado de que la forma más fácil de agregar una función es agregar nuevo código. Modificar el código existente para manejar nuevas situaciones a menudo es difícil porque ese código no es ni bien comprendido ni bien documentado. A medida que el tamaño de un programa aumenta, a veces en uno o dos órdenes de magnitud durante varios años, los cambios se vuelven más difíciles de varias maneras. En primer lugar, hay más código para cambiar; un cambio que podría haberse hecho en una o dos partes del programa original ahora requiere modificar muchas secciones del código. En segundo lugar, es más difícil encontrar las rutinas que deben cambiarse. Como resultado, los propietarios no pueden agregar nuevas funciones lo suficientemente rápido. Los clientes pueden cambiar a un producto más joven para obtener esas características. La empresa experimenta una notable disminución en los ingresos; cuando lanzan una nueva versión, suscita interés en una base de clientes cada vez más reducida. Si intentan mantenerse al día con el mercado aumentando su fuerza laboral, los costos adicionales de los cambios y las demoras conducen a una mayor pérdida de clientes.

* 1. Reducción de rendimiento

A medida que el tamaño del programa crece, exige más memoria de la computadora y hay más demoras mientras el código debe intercambiarse desde el almacenamiento masivo. El programa responde más lentamente; los clientes deben actualizar sus computadoras para obtener una respuesta aceptable. La eficiencia también disminuye debido a un diseño deficiente. El software ya no se comprende bien y los cambios pueden afectar negativamente el rendimiento. Un producto más joven, cuyo diseño original reflejaba la necesidad de características recientemente introducidas, funcionará más rápido o usará menos memoria.

* 1. Disminución de la confiabilidad

A medida que se mantiene el software, se introducen errores. Incluso en los primeros años de la industria, los observadores pudieron documentar situaciones en las que cada error corregido introducía (en promedio) más de un error. Cada vez que se intentaba disminuir la tasa de fallos de los sistemas, empeoraba. A menudo, la única opción era abandonar el producto o al menos dejar de reparar errores. Me han contado sobre productos de software más antiguos en los que la lista de errores conocidos pero aún no reparados superaba las 2000 entradas.

1. Reducción de los costos del envejecimiento del software

Los programadores inexpertos a menudo pueden ser reconocidos por la alegría que muestran la primera vez que obtienen resultados correctos de un programa. "¡He terminado; ¡funciona!" es el grito de un nuevo programador que acaba de tener una exitosa primera demostración. El programador experimentado se da cuenta de que esto es solo el comienzo. Saben que cualquier producto serio requiere pruebas extensas, revisión y revisión después de la primera ejecución exitosa. El trabajo invertido por organizaciones responsables y profesionales después de la primera ejecución exitosa y antes del primer lanzamiento suele ser mucho mayor que el necesario para obtener la primera ejecución exitosa. Sin embargo, incluso los programadores experimentados se centran en ese primer lanzamiento. Nuestra experiencia con el envejecimiento del software nos dice que debemos mirar mucho más allá del primer lanzamiento hasta el momento en que el producto sea antiguo.

Demasiados documentos en conferencias de ingeniería de software se centran en los problemas para llegar al primer lanzamiento. Demasiados documentos se centran en los problemas de gestión (por ejemplo, gestión y control de configuraciones). Lidiar con el envejecimiento del software requiere más que "gestión paciente"; requiere una ingeniería sólida. El propósito del resto de este documento es considerar qué acciones podríamos tomar para reducir los costos asociados con el envejecimiento del software.

1. Medicina preventiva

Dado que el envejecimiento del software es un problema tan serio, la primera pregunta que debemos hacernos es qué podemos hacer para retrasar el deterioro y limitar sus efectos.

* 1. Diseño para el éxito

El primer paso para controlar el envejecimiento del software es aplicar el antiguo lema "diseñar para el cambio". Desde principios de los años 70, sabemos cómo diseñar software para el cambio. El principio a aplicar se conoce con varios nombres, como "ocultamiento de información", "abstracción", "separación de preocupaciones", "ocultamiento de datos" o, más recientemente, "orientación a objetos". Para aplicar este principio, se comienza por tratar de caracterizar los cambios que es probable que ocurran a lo largo de la "vida útil" del producto. Dado que no podemos prever los cambios reales, las predicciones serán sobre clases de cambios, por ejemplo, representación de expresiones revisada, reemplazo del terminal por un nuevo tipo, cambios en la interfaz de usuario, cambio a un nuevo sistema de ventanas, etc. Dado que es imposible hacer que todo sea igualmente fácil de cambiar, es importante estimar las probabilidades de cada tipo de cambio. Luego, se organiza el software de manera que los elementos más propensos a cambiar están "confinados" a una pequeña cantidad de código, de modo que si esas cosas cambian, solo se vería afectada una pequeña cantidad de código. A pesar de la simplicidad de este principio y a pesar de su amplia aceptación, no veo mucho software que esté bien diseñado desde este punto de vista. Vale la pena examinar algunas de las razones por las cuales la industria no aplica este principio correctamente.

Muchos manuales de software mencionan esta técnica, pero la cubren de manera superficial. Dicen que uno debería ocultar o abstraer los "detalles de implementación", pero no discuten ni ilustran el proceso de estimar la probabilidad de cambio para diversas clases de cambios. El principio es simple; aplicarlo correctamente requiere mucho pensamiento sobre la aplicación y el entorno. Los libros de texto no dejan eso claro.

Muchos programadores son impacientes con tales consideraciones; están tan ansiosos por hacer funcionar la primera versión o cumplir con algún plazo inminente que no se toman el tiempo para diseñar pensando en el cambio. La gestión está tan preocupada por el próximo plazo (y tan ansiosa por ascender) que los costos futuros de mantenimiento no tienen la máxima prioridad.

Los diseños que resultan de una aplicación cuidadosa del ocultamiento de información son bastante diferentes de los diseños "naturales" que resultan del trabajo intuitivo de la mayoría de los programadores. La intuición del programador es pensar en pasos en el procesamiento de datos, no en cambios probables. Incluso cuando se les dice que asocien cada módulo con un "secreto", algo que es probable que cambie y que debería encapsularse, usan "secretos" del tipo "cómo...". Los diseñadores tienden a imitar otros diseños que han visto. No ven muchas aplicaciones efectivas del ocultamiento de información. Un ejemplo de diseño basado en el ocultamiento de información se encuentra en [9].

Los programadores tienden a confundir principios de diseño con lenguajes. Por ejemplo, creen que no se pueden aplicar ideas "orientadas a objetos" sin un lenguaje "orientado a objetos". Peor aún, piensan que han aplicado las técnicas si han utilizado ese lenguaje.

Muchas personas que se dedican al desarrollo de software no tienen una educación adecuada para el trabajo. Temas que son "pan comido" para aquellos que asisten a esta conferencia son desconocidos o términos vagos para muchos que escriben software. Cada industria tiene sus propias conferencias de software y muchos programadores en cada industria trabajan como si sus problemas fueran únicos.

Los investigadores en Ingeniería de Software continúan predicando a los convertidos, escribiendo papers entre ellos y ignorando lo que está sucediendo donde se escribe la mayoría del software. Asumen que "diseñar para el cambio" es un problema antiguo, no uno que requiera más trabajo. ¡Están equivocados!

Aunque el principio del ocultamiento de información se enunció por primera vez a principios de los años 70 (e incluso se ilustró antes), es raro encontrar un producto de software que haya sido diseñado correctamente desde este punto de vista. El código a menudo es ingenioso, eficiente y correcto; realiza funciones bastante asombrosas, pero rara vez se diseña para cambiar fácilmente. El problema no es que nadie sepa cómo hacerlo, sino que la mayoría de los programadores no lo hacen. Sospecho que algunos programadores piensan que su programa será tan bueno que no tendrá que cambiarse. Esto es absurdo. Los únicos programas que no se cambian son aquellos tan malos que nadie quiere usarlos. Diseñar para el cambio es diseñar para el éxito.

* 1. Mantener registros - documentación

Incluso cuando el código está diseñado de manera que los cambios se pueden llevar a cabo eficientemente, los principios de diseño y las decisiones de diseño a menudo no se registran de una forma útil para los futuros mantenedores. La documentación es el aspecto de la ingeniería de software más descuidado tanto por los investigadores académicos como por los profesionales. Es común escuchar a un programador decir que el código es su propia documentación; incluso investigadores de lenguajes altamente respetados sostienen esta posición, argumentando que si se utiliza su último lenguaje, la estructura será explícita y obvia.

Cuando se escribe documentación, suele estar mal organizada, incompleta e imprecisa. A menudo, la cobertura es aleatoria; un programador o gerente decide que una idea en particular es ingeniosa y escribe un memo al respecto, mientras que otros temas igualmente importantes son ignorados. En otras situaciones, donde la documentación es un requisito contractual, se contrata a un redactor técnico, que no comprende el sistema, para escribir la documentación. La documentación resultante es ignorada por los programadores de mantenimiento porque no es precisa. Algunos proyectos mantienen dos juegos de libros; está la documentación oficial, escrita según lo requerido por el contrato, y la documentación real, escrita informalmente cuando surgen problemas específicos.

La documentación que parece clara y adecuada para sus autores a menudo es tan clara como el barro para el programador que debe mantener el código 6 meses o 6 años después. Incluso cuando la información está presente, el programador de mantenimiento no sabe dónde buscarla. Es casi tan común encontrar que el mismo tema se cubre dos veces, pero que las declaraciones en la documentación son inconsistentes entre sí y con el código.

La documentación no es un tema de investigación "atractivo". El año pasado, sugerí a un líder de un proyecto de Esprit que estaba buscando un tema para una conferencia, que se centrara en la documentación. Su respuesta fue que no sería interesante. Objeté, diciendo que hay muchos aspectos interesantes en este tema. Su respuesta fue que el problema no era que la discusión no fuera interesante, sino que el tema no sonaría interesante y no atraería a la audiencia.

Durante los últimos cinco o seis años, mi propia investigación y la de muchos de mis estudiantes y colegas cercanos se ha centrado en los problemas de la documentación. Hemos demostrado cómo los métodos matemáticos pueden usarse para proporcionar documentación clara, concisa y sistemática del diseño del programa [3,4]. Hemos inventado e ilustrado nuevas notaciones matemáticas que son mucho más adecuadas para su uso en documentación, pero no menos formales [5,6,7]. La reacción de académicos y profesionales a este trabajo ha sido provocadora de reflexiones. Ambos lados no reconocen la documentación como el tema de nuestro trabajo. Los académicos siguen señalando que estamos descuidando "obligaciones de prueba"; los revisores industriales clasifican nuestro trabajo como "verificación", que consideran (a menudo correctamente) demasiado difícil y teórico. Ningún grupo puede ver la documentación como un tema más fácil y, en cierto sentido, más importante que la verificación. Para ellos, la documentación es ese "blah blah" que hay que escribir. De hecho, a menos que podamos resolver el problema de la documentación, el trabajo de verificación será una pérdida de tiempo.

Al hablar con personas que desarrollan software comercial, encontramos que la documentación se descuida porque no acelerará el próximo lanzamiento. Nuevamente, programadores y gerentes están tan impulsados por el plazo más inminente que no pueden planificar para la vejez del software. Si reconocemos que el envejecimiento del software es inevitable y costoso, que el primer o próximo lanzamiento del programa no es el final de su desarrollo, que los costos a largo plazo superarán las ganancias a corto plazo, comenzaremos a tomar la documentación más en serio.

Cuando comenzamos a tomarnos en serio la documentación, veremos que, al igual que en otros tipos de ingeniería, la documentación de software debe basarse en matemáticas. Cada documento será una representación de una o más relaciones matemáticas. La única forma práctica de registrar la información necesaria en una documentación adecuada será mediante el uso de notación formalmente definida.

* 1. Segundas opiniones - revisiones

En ingeniería, al igual que en medicina, la necesidad de revisiones por parte de otros profesionales nunca se cuestiona. En el diseño de un edificio, un barco o una aeronave, siempre hay una serie de documentos de diseño cada vez más precisos y cada uno es revisado cuidadosamente por otros. Aunque el tema de las revisiones de diseño se discute ampliamente en las conferencias de ingeniería de software, es sorprendente ver cuántas veces se producen programas comerciales sin una revisión adecuada. Hay muchas razones para esto:

Muchos programadores no tienen formación profesional en software en absoluto. Algunos son ingenieros de otros campos, algunos son "científicos caídos" que aprendieron programación incidentalmente mientras obtenían su educación. Algunos eran matemáticos y otros provenían de antecedentes no técnicos. En muchas de esas áreas, el concepto de preparar y llevar a cabo una revisión de diseño no existe.

Incluso entre aquellos que tienen títulos en Ciencias de la Computación, han tenido una educación que ha descuidado preocupaciones profesionales como la necesidad de documentación y revisiones de diseño. El énfasis está en las matemáticas y la ciencia; la disciplina profesional no es un tema para una educación "liberal".

La mayoría de los profesionales (y muchos investigadores) no saben cómo proporcionar documentación precisa y legible de un diseño, distinta de una implementación. No hay una descripción precisa, aparte del código detallado, disponible para su revisión. Las revisiones de diseño al principio de un proyecto, cuando serían más útiles, se reducen a sesiones de charla porque no hay documentos de diseño detallados para discutir.

Gran parte del software se produce como una industria casera, donde no hay personas que puedan servir como revisores calificados y no hay financiamiento para contratar revisores externos.

Con frecuencia, el software se produce bajo presión de tiempo que engaña a los diseñadores haciéndoles pensar que no tienen tiempo para revisiones adecuadas.

Muchos programadores consideran la programación como un "arte" y se molestan con la idea de que alguien podría o debería revisar el trabajo que han hecho. He conocido a programadores que dejan de trabajar porque se molestan por el hecho de que su trabajo estaría sujeto a revisión.

Para cualquier organización que tenga la intención de mantener sus productos de software durante varios años, las revisiones son esenciales y deben tomarse más en serio de lo que es común ahora. En particular, para mitigar los problemas del envejecimiento del software, cada diseño debe ser revisado y aprobado por alguien cuyas responsabilidades sean para el futuro a largo plazo del producto. Las revisiones realizadas por personas preocupadas por el mantenimiento deben llevarse a cabo cuando se propone el diseño y mucho antes de que exista código. Puedes encontrar una discusión sobre cómo revisar documentos de diseño en [2].

* 1. Por qué el envejecimiento del software es inevitable

Incluso si tomamos todas las medidas preventivas razonables y lo hacemos de manera religiosa, el envejecimiento es inevitable. Nuestra capacidad para diseñar para el cambio depende de nuestra capacidad para predecir el futuro. Solo podemos hacerlo aproximadamente y de manera imperfecta. A lo largo de los años, realizaremos cambios que violan nuestras suposiciones originales. La documentación, incluso si es formal y precisa, nunca será perfecta. Las revisiones sacarán a la luz problemas que los diseñadores pasan por alto, pero también hay problemas que los revisores pasan por alto. Las medidas preventivas valen la pena, pero cualquiera que piense que esto eliminará el envejecimiento está viviendo en un mundo de fantasía.

1. Geriatría del software

La prevención siempre es la mejor medicina, pero aún tenemos que lidiar con el software antiguo. Esta sección describe varias cosas que se pueden hacer para tratar el envejecimiento del software que ya ha ocurrido.

* 1. Detener el deterioro

Si el software ha sido mantenido durante algún tiempo sin mucha atención a los problemas planteados aquí, se observará un marcado deterioro. El primer paso debería ser ralentizar el progreso del deterioro. Esto se logra introduciendo o recreando estructura cada vez que se realizan cambios. Los principios de diseño mencionados anteriormente pueden usarse para guiar el cambio y el mantenimiento también. Si se cambia una decisión de diseño sobre el sistema, la nueva estructura de datos o algoritmo puede ocultarse (encapsularse) de manera que facilite cualquier cambio futuro en ese aspecto del sistema. Revisiones cuidadosas deben asegurar que cada cambio sea consistente con la intención de los diseñadores originales, que el concepto de diseño original no sea violado por los nuevos cambios.

Detener el deterioro es, como muchas otras cosas en Ingeniería de Software, más fácil de decir que de hacer. Muchas empresas han permitido que el crecimiento canceroso continúe sin control en su software durante años. Cuando las cosas van bien, el crecimiento es rápido y no hay una razón evidente para ser cauteloso. El resultado es que un solo proyecto puede existir en muchas versiones, cada una con estructuras ligeramente diferentes y basadas en suposiciones ligeramente diferentes. Cuando el período de rápido crecimiento ha terminado, cada cambio debe hacerse muchas veces y los mantenedores se confunden por la profusión de versiones casi iguales. Alguien tiene que realizar un estudio serio de todas esas versiones y registrar las diferencias. Luego, un equipo tendrá que ponerse de acuerdo sobre la estructura adecuada y todas las versiones tendrán que ajustarse a ese molde. En un momento en que las cosas no van bien, es difícil obtener suficiente personal para hacer el trabajo adecuadamente. Deben crearse y revisarse nuevos documentos. Luego, el código debe verificarse para asegurarse de que se haya vuelto consistente con estos nuevos documentos. Un proceso así podría llevar varios años y durante ese tiempo seguirán llegando demandas de cambios y correcciones. Cortar el crecimiento desde el principio es mucho más preferible. La retirada siempre es dolorosa.

* 1. Documentación retrospectiva

Un paso importante para frenar el envejecimiento del software más antiguo, y a menudo rejuvenecerlo, es mejorar la calidad de la documentación. A menudo, la documentación es descuidada por los programadores de mantenimiento debido a su prisa por corregir problemas informados por los clientes o para introducir características demandadas por el mercado. Cuando documentan su trabajo, a menudo lo hacen mediante un memorando que no está integrado en la documentación existente, sino que simplemente se agrega a ella. Si el software es realmente valioso, la documentación no estructurada resultante puede, y debería, ser reemplazada por una documentación cuidadosamente estructurada que haya sido revisada para ser completa y correcta. A menudo, cuando se sugiere tal proyecto, los programadores (que rara vez están entusiasmados con cualquier forma de documentación) ridiculizan la sugerencia como impráctica. Sus intereses son a corto plazo y su satisfacción laboral proviene de ejecutar programas. No obstante, hay situaciones en las que es de interés del propietario insistir en que el producto esté documentado en una forma que pueda servir como referencia para futuros programadores de mantenimiento.

Un efecto secundario agradable de los esfuerzos de documentación es a menudo la mejora del software. La documentación formal que recomendamos requiere un examen detallado y sistemático del código y a menudo revela errores, funciones duplicadas o casi idénticas y formas de mejorar el rendimiento. En un experimento reciente, le pedí a un estudiante de pregrado que produjera documentación para un software que ya no funcionaba. El autor había dejado nuestro país. Aunque no se le pidió que encontrara errores, el análisis sistemático necesario para crear la documentación formal lo obligó a examinar cada rutina cuidadosamente. Sugería algunos cambios y ahora el software es funcional, además de estar bien documentado para cambios futuros.

* 1. Modularización incremental retrospectiva

Aunque todos los expertos en software ahora admiten la importancia de la modularización y la mayoría de los programas grandes tienen algunas unidades consideradas módulos, una buena comprensión de los principios de la modularización rara vez se refleja en el código. La modularización requiere más que simplemente identificar subrutinas o pequeños grupos de procedimientos y llamarlos módulos. Cada módulo debe comprender todos los programas "conocen" (están basadas en) una decisión de diseño particular que es probable que cambie. Reconocer cosas que probablemente cambiarán requiere experiencia, y ocultar o confinar con éxito el conocimiento de una decisión de diseño a un módulo requiere habilidades y comprensión que son raras. Aún así, los programadores que comprenden el ocultamiento de información y la abstracción generalmente pueden encontrar segmentos de código que deberían ser módulos y recopilarlos en unidades. Un consultor, que ve el software con ojos frescos, a menudo puede mostrar cómo se realiza el trabajo. Hacerlo facilita en gran medida el mantenimiento futuro del código. A menudo, estas mejoras se pueden hacer a un costo mínimo como un efecto secundario de los cambios que deben hacerse de todos modos.

* 1. Amputación

Ocasionalmente, una sección de código ha sido modificada tantas veces y de manera tan descuidada que no vale la pena guardarla. Grandes secciones pueden descartarse y reemplazarse por "muñones" artificiales que realizan la función de alguna otra manera. La amputación siempre es una decisión difícil y controvertida. Aquellos que han creado el código antiguo no están dispuestos a admitir que no vale la pena conservarlo. Nuevamente, los consultores suelen ser útiles si se les puede informar completamente. No tienen el apego emocional al código que los autores podrían tener.

* 1. Cirugía mayor: reestructuración

Cuando una familia grande e importante de productos se sale de control, es apropiado realizar un esfuerzo importante para reestructurarla. El primer paso debe ser reducir el tamaño de la familia de programas. Se deben examinar las diversas versiones para determinar por qué y cómo difieren. Si se pueden introducir módulos que oculten esas diferencias, acordar (y documentar) interfaces estándar para esos módulos y luego realizar esos cambios en las diversas versiones, se pueden colapsar las versiones en un solo sistema que difiera solo en algunos módulos. Reemplazar las versiones antiguas con las reestructuradas permite que los cambios futuros en el código compartido sean compartidos por muchas versiones. En muchas situaciones, las versiones separadas pueden combinarse en una sola mediante la introducción de "interruptores" que se verifican en tiempo de ejecución para determinar qué versión de comportamiento se desea. Esto introduce una pequeña ineficiencia en tiempo de ejecución pero reduce en gran medida el tamaño del personal de mantenimiento. He visto algunas organizaciones que pudieron ofrecer lo que parecía ser una amplia familia de productos distribuyendo un solo fragmento de código y configurando interruptores ocultos para crear sistemas que parecían bastante diferentes. Los costos de mantenimiento de estas organizaciones son mucho más bajos de lo que serían si tuvieran versiones separadas. Lamentablemente, algunos de sus clientes encontraron los interruptores y pudieron disfrutar de los beneficios de funciones que no habían comprado. A pesar de esto, sospecho que el fabricante de software estaba adelante debido a los costos reducidos de mantenimiento.

1. Planificación anticipada

Si queremos prevenir, o al menos frenar, el envejecimiento del software, debemos reconocerlo como un problema y planificar para ello. Cuanto antes planifiquemos para la vejez, más podremos hacer.

* 1. Un nuevo "Estilo de vida"

Es hora de dejar de actuar como si "hacer que funcione" fuera lo único que importa. Obviamente, es importante llevar un producto al cliente rápidamente, pero no podemos seguir actuando como si no hubiera un mañana. No debemos permitir que las presiones de hoy resulten en un producto (y empresa) mutilado el próximo año. No podemos hacer un buen trabajo bajo estrés, especialmente el estrés constante de una crisis de 25 años. La industria misma debe tomar medidas para frenar el rápido ritmo de desarrollo. Esto se puede lograr imponiendo estándares en la estructura y la documentación, asegurándose de que los productos producidos utilizando "atajos" no lleven el "sello de calidad" de la industria.

* 1. Planificación para el cambio

Los diseños deben documentarse y revisarse cuidadosamente antes de que comience la codificación. Los programas deben documentarse y revisarse. Los cambios deben documentarse y revisarse. Un análisis exhaustivo de los cambios futuros debe ser parte de cada diseño de producto y acción de mantenimiento. Las organizaciones que son más grandes que unas pocas personas deberían tener un profesional o un departamento dedicado a revisar los diseños para la capacidad de cambio. Deberían tener el poder de vetar cambios que hagan que las cosas se hagan más rápido ahora pero a un gran costo más tarde.

* 1. Si no está documentado, no está hecho

Si un producto no está documentado según su diseño, utilizando la documentación como un medio de diseño [1], ahorraremos un poco hoy, pero pagaremos mucho más en el futuro. Es mucho más difícil recrear la documentación de diseño que crearla a medida que avanzamos. La documentación que se ha creado después de que se haya completado el diseño y se haya enviado el producto generalmente no es muy precisa. Además, dicha documentación no estaba disponible cuando (y si) se revisó el diseño antes de la codificación. Como resultado, incluso si la documentación es tan buena como lo habría sido, ha costado más y ha sido menos valiosa.

Ingenieros eléctricos, el software en el campo de la fabricación automatizada es escrito por ingenieros mecánicos, etc. Los ingenieros de programación en esas industrias no se consideran a sí mismos como profesionales en el sentido en que lo hacen los ingenieros aeronáuticos o nucleares. Además, no han recibido una educación formal en el campo en el que ahora trabajan. Descubrimos que los ingenieros que escriben programas saben muy poco sobre la ciencia de la computación, pero los graduados en ciencias de la computación saben muy poco sobre los procedimientos y disciplinas de ingeniería. A menudo escucho: "cualquiera puede escribir un programa" y es cierto, pero los programas escritos de manera no profesional envejecerán mucho más rápidamente que los programas escritos por ingenieros que han recibido educación en las matemáticas y técnicas importantes para el diseño de programas, [8].

* 1. Planes de ahorro para la jubilación

En otras áreas de la ingeniería, se reconoce la obsolescencia del producto y se incluye en los planes de diseño y marketing. El automóvil nuevo que compras hoy ya es "antiguo" para los ingenieros que ya están trabajando en modelos futuros. El automóvil tiene garantía solo por un tiempo (muy) limitado y las piezas de repuesto también deben estar disponibles solo durante períodos establecidos. Cuando compramos un automóvil, sabemos que envejecerá y eventualmente tendrá que ser reemplazado. Si somos sabios, comenzamos a planificar ese reemplazo tanto financieramente como leyendo sobre nuevos desarrollos. Los fabricantes muestran una previsión similar. Solo en la industria del software, la gente trabaja como si su producto "viviera" para siempre. Cada diseñador y comprador de software debería estar planeando el día en que el producto deba ser reemplazado. Una parte de esta planificación es la planificación financiera, asegurándose de que cuando llegue el momento de instalar o desarrollar un nuevo producto, los fondos y las personas estén disponibles.

1. Barreras para el progreso

Si vamos a mejorar el problema del envejecimiento del software, tendremos que hacer algunos cambios profundos en nuestra profesión. Hay cuatro barreras básicas para el progreso en Ingeniería de Software. Estas son actitudes y suposiciones que hacen imposible que la investigación marque la diferencia.

* 1. ¿Una crisis de 25 años?

Escuché por primera vez el término "crisis del software" hace 25 años y lo he escuchado usar para describir un problema actual todos los años desde entonces. Esto es claramente absurdo. Una crisis es una emergencia seria, repentina y a corto plazo. La llamada "crisis del software" ciertamente es seria, pero no es ni repentina ni a corto plazo. No puede tratarse como si fuera una emergencia repentina: necesita una terapia cuidadosa a largo plazo. Las soluciones "rápidas y fáciles" nunca han funcionado y no funcionarán en el futuro. La frase "crisis del software" ayuda a tratar con ciertas agencias de financiamiento, pero impide el análisis profundo necesario para curar una enfermedad crónica. Conduce a un pensamiento a corto plazo y a un envejecimiento rápido del software.

* 1. "Nuestra industria es diferente."

El software se utiliza en casi todas las industrias, como la aeronáutica, militar, automotriz, energía nuclear y telecomunicaciones. Cada una de estas industrias se desarrolló como una comunidad intelectual antes de depender del software. Cada una tiene sus propias organizaciones profesionales, organizaciones comerciales, sociedades técnicas y revistas técnicas. Como resultado, encontramos que muchas de estas industrias están abordando sus problemas de software sin estar al tanto de los esfuerzos en otras industrias. Cada industria ha desarrollado su propio vocabulario y documentos que describen la forma en que se debe construir el software. Algunas han desarrollado sus propias notaciones de especificación y convenciones de diagramación. Hay muy poca comunicación cruzada. Los ingenieros de la industria nuclear discuten sus problemas de software en reuniones de la industria nuclear, mientras que los ingenieros de telecomunicaciones discuten problemas muy similares en reuniones completamente diferentes. Para llegar a su audiencia prevista, un artículo sobre ingeniería de software tendrá que publicarse en muchos lugares diferentes. Nadie quiere hacer eso (pero los comités de promoción lo recompensan).

Este aislamiento intelectual es inapropiado y costoso. Es inapropiado porque los problemas son muy similares. A veces, las estructuras de costos que afectan a las soluciones son diferentes, pero los problemas técnicos son muy similares. Es costoso porque el aislamiento a menudo resulta en que las personas reinventen la rueda, y con mayor frecuencia, reinventen ruedas muy irregulares y estructuralmente débiles. Por ejemplo, la industria de las telecomunicaciones y aquellos interesados en los sistemas de fabricación rara vez se comunican, pero sus problemas de protocolo de comunicación tienen muchas similitudes. Se observa que las personas que trabajan en las dos industrias a menudo no se dan cuenta de que tienen los mismos problemas y repiten los errores del otro. Incluso la separación entre software crítico para la seguridad y software no crítico para la seguridad (que podría parecer sensata) es desafortunada porque las ideas que funcionan bien en una situación a menudo son aplicables en las demás.

Necesitamos construir una identidad profesional que se extienda a personas en todas las industrias. En este momento, alcanzamos a algunas personas en todas las industrias, pero no parece que estemos llegando a la persona típica en esas industrias.

* 1. ¿Dónde están los profesionales?

La partición de personas e industrias con problemas de software es un síntoma de un problema diferente. Aunque tenemos muchas personas que reciben un salario por escribir software, no tenemos ingenieros de software en el sentido en que tenemos ingenieros aeronáuticos, eléctricos o civiles. Estos últimos grupos son principalmente personas que han recibido una educación profesional en el campo en el que trabajan, pertenecen a sociedades profesionales en ese campo y se espera que se mantengan al día en ese campo. En cambio, descubrimos que el software en el campo nuclear es escrito por ingenieros nucleares que han aprendido un lenguaje de programación, el software en el campo de las telecomunicaciones es escrito por comunicaciones

* 1. Hablándonos a nosotros mismos

Los investigadores deben empezar a replantearse su audiencia. Con demasiada frecuencia, escribimos artículos para impresionar a nuestros colegas, a otros investigadores. Aún peor, si intentamos escribir un artículo para el profesional, los árbitros se quejan si incluimos definiciones o problemas básicos. Terminamos escribiendo artículos que son leídos por nuestros colegas investigadores, pero no por muchos otros. También dedicamos muy poco tiempo a descubrir lo que los profesionales saben, piensan y necesitan. En las Facultades de Ingeniería, se reconoce la práctica profesional como esencial para una buena enseñanza e investigación. En muchas facultades de Ciencias, se ve simplemente como una forma de ganar dinero extra. Esta es una de las muchas razones por las que creo que los Departamentos de Ciencias de la Computación funcionarían mejor si siempre fueran parte de una Facultad de Ingeniería.

1. Conclusiones para nuestra profesión

(1) No podemos asumir que lo antiguo se conoce y no funcionó. Si no funcionó, tenemos que averiguar por qué. A menudo es porque no se intentó.

(2) No podemos asumir que lo antiguo funcionará. A veces, las creencias ampliamente aceptadas son incorrectas.

(3) No podemos ignorar los pequeños grupos de ingeniería de software. Juntos superan en número a las personas que leerán nuestros artículos o vendrán a nuestras conferencias.

(4) Los productos modelo son imprescindibles. Si no podemos ilustrar un principio con un producto real, puede que haya algo mal con el principio. Incluso si el principio es correcto, sin modelos reales, la tecnología no se transferirá. Los profesionales imitan lo que ven en otros productos. Si queremos que nuestras ideas se difundan, tenemos que ponerlas en productos. Hay un lugar legítimo, honorable e importante para los investigadores que "no inventan nuevas ideas, sino que aplican, demuestran y evalúan las antiguas.

(5) Necesitamos que la frase "ingeniero de software" signifique algo. Hasta que tengamos estándares profesionales, requisitos educativos razonablemente estandarizados y una identidad profesional, no tenemos derecho a usar la frase "Ingeniería de Software".

1. Referencias

[1] HESTER, S.D., PARNAS, D.L., UTTER, D. F., “Using Documentation as a Software Design Medium”, Bell System Technical Journal, 60, 8, October 1981, pp. 1941-1977.

[2] PARNAS, D. L., “Active Reviews: Principles the 8th International Engineering, London, August 1985. Also published in Journal of System and Software, December 1987.

[3] VAN SCHOUWEB, A.J., PARTNAS, D.L., MADEY, J. “Documentation Requirements for Computer Systems”, presented at RE ’93 IEEE International Simposium on Reqirements Engineering, San Diego, CA, 4-6 Janauary, 1993.

[4] PARNAS, D.L., MADEY, J. “Functionalk Documentation for Computer Systems Engineering (Version 2)”, CRL Report 237, CRL-TRIO McMaster University, September 1991, 14 pgs. (to be published in Science Computer Programming”)

[5] PARNAS, D.L., “Tabular Representation of Relations”CRL Report 260, CRL. McMaster University, October 1992, 17 pgs.

[6] JANICKI, R., “Towards a Formal Semantics of Tables”, CRL Report 264. CRL McMaster University September 1993, 18 pgs.

[7] ZUCKER, J.I., “Norman and Invented Function Tables”, CRL Report 265, CRL, McMaster University, December 1993, 16 pgs.

[8] PARNAS D.L., “Education for computing Professionals”, IEEE Computer, vol. 23, no. 1, January 1990, pp. 17-22.

[9] PARNAS, D.L. CLEMENTS, P.C., WEISS, D.M., “The Modular Structure of Complex Systems”, IEEE Transactions on Software Engineering, March 1985, Vol. SE-111 No. 3, pp. 259-266. Also published in Proceedings of 7th International Conference on Software Engineering, March 1984, pp. 408-417.

Créditos:

David Lorge Parnas
Communications Research Laboratory
Deparment of Electrical and Computer Engineering
MacMaster University, Hamilton, Ontario, Canada L8S 4K1

Fuente: https://www.researchgate.net/publication/3560773\_Software\_aging