LIBROSGRATIS.DEV

97 cosas que todo programador debería saber

97 consejos traducidos al español

EDITOR ORIGINAL: KEVLIN HENNEY

Atribución y licencia

Traducción al español a partir de las contribuciones originales listadas en Programmer 97-things . La obra original indica: “This work is licensed under a Creative Commons Attribution 3” (Creative Commons Attribution 3.0).
Esta versión forma parte del repositorio midudev/libros-programacion-gratis .

Índice

Actúa con prudencia
Seb Rose

Averigua qué haría el usuario (tú no eres el usuario)
Giles Colborne

La belleza está en la simplicidad
Jørn Ølmheim

Ten cuidado al compartir
Udi Dahan

Primero revisa tu código antes de buscar culpar a otros
Allan Kelly

Codifica en el lenguaje del dominio
Dan North

El diseño del código sí importa
Steve Freeman

Codificando con la razón
Yechiel Kimchi

Comenta sólo lo que el código no dice
Kevlin Henney

La conveniencia no es una -bilidad
Gregor Hohpe

Distingue excepciones de Negocio de las excepciones Técnicas
Dan Bergh Johnsson

Lenguajes Específicos del Dominio (DSL)
Michael Hunger

No seas lindo con tus datos de prueba
Rod Begbie

No sólo aprendas el lenguaje, entiende su cultura
Anders Norås

No confíes en el "Aquí sucede la magia"
AlanGriffiths

¡No toques ese código!
Cal Evans

Los números de punto flotante no son reales
Chuck Allison

La regla de oro del diseño de API
Michael Feathers

El trabajo duro no paga
Olve Maudal

Mejora el código quitándolo
Pete Goodliffe

La comunicación entre procesos afecta el tiempo de respuesta de la aplicación
Randy Stafford

Aprende a usar las herramientas de línea de comandos
Carroll Robinson

Conoce tu IDE
Heinz Kabutz

Conoce tu próximo Commit
Dan Bergh Johnsson

Aplica los principios de la programación funcional
Edward Garson

Automatiza el estándar de codificación
Filip van Laenen

Antes de Refactorizar
Rajith Attapattu

La Regla Boy Scout
Uncle Bob

Escoge tus herramientas con cuidado
Giovanni Asproni

El código es diseño
Ryan Brush

Revisiones de código
Mattias Karlsson

Un comentario acerca de los comentarios
Cal Evans

Aprendiendo continuamente
Clint Shank

Implementa rápido y con frecuencia
Steve Berczuk

Haz mucha práctica deliberada
Jon Jagger

No tengas miedo de romper cosas
Mike Lewis

¡No ignores ese error!
Pete Goodliffe

No claves tu programa en la posición vertical
Verity Stob

No te repitas
Steve Smith

Encapsula Comportamiento, no sólo Estado
Einar Landre

Cumple tus ambiciones con Software Libre
Richard Monson-Haefel

El mito del Gurú
Ryan Brush

¿Cómo usar un Gestor de Errores?
Matt Doar

Instalame
Marcus Baker

Mantén limpia la compilación
Johannes Brodwall

Conoce bien más de dos lenguajes de programación
Russel Winder

Conoce tus límites
Greg Colvin

Los grandes datos interconectados pertenecen a una base de datos
Diomidis Spinellis

Aprende un lenguaje extranjero
Klaus Marquardt

Aprende a decir "Hola, Mundo"
Thomas Guest

El linker no es un programa mágico
Walter Bright

Haz las Interfaces fáciles de usar correctamente y difíciles de usar incorrectamente
Scott Meyers

El paso de mensajes lleva a una mejor escalabilidad en sistemas paralelos
Russel Winder

Oportunidades perdidas del Polimorfismo
Kirk Pepperdine

Un binario
Steve Freeman

Adueñate (y Refactoriza) la compilación
Steve Berczuk

Da preferencia a tipos de Dominio Específico que los tipos primitivos
Einar Landre

El Programador Profesional
Uncle Bob

Suelta el ratón y aléjate del teclado
Cay Horstmann

Lee las humanidades
Keith Braithwaite

Resiste la tentación del patrón Singleton
Sam Saariste

La Simplicidad viene de la Reducción
Paul W. Homer

Inicia con un Sí
Alex Miller

Toma ventaja de las herramientas de análisis de código
Sarah Mount

Prueba precisa y concretamente
Kevlin Henney

Las pruebas son el rigor ingenieril del desarrollo de software
Neal Ford

Dos cabezas son a menudo mejores que una
Adrian Wible

Codificación Ubuntu para tus amigos
Aslam Khan

Usa el algoritmo y estructura de datos correctos
JC van Winkel

El WET dispersa los cuellos de botella en el rendimiento
Kirk Pepperdine

Escribe código como si tuvieras que mantenerlo por el resto de tu vida
Yuriy Zubarev

Escribe las pruebas para las personas
Gerard Meszaros

Tus clientes no quieren decir lo que dicen

Aprende a hacer estimaciones
Giovanni Asproni

Deja que tu proyecto hable por sí mismo
Daniel Lindner

La longevidad de las soluciones provisionales
Klaus Marquardt

Haz lo invisible más visible
Jon Jagger

Mensaje al futuro
Linda Rising

Noticias raras - Los testers son tus amigos
Burk Hufnagel

Sólo el código dice la verdad
Peter Sommerlad

Programa en pareja y siente el flujo
Gudny Hauknes, Ann Katrin Gagnat, y Kari Røssland

Evita errores
Giles Colborne

Pon todo bajo Control de Versiones
Diomidis Spinellis

Lee el código
Karianne Berg

Reinventa la rueda frecuentemente
Jason P Sage

El camino al mejor rendimiento está lleno de sucias bombas de código
Kirk Pepperdine

El Principio de Responsabilidad Única
Uncle Bob

Retrocede y Automatiza, Automatiza, Automatiza
Cay Horstmann

Prueba el comportamiento requerido, no el comportamiento incidental
Kevlin Henney

Haz pruebas mientras duermes (y los fines de semana)
Rajith Attapattu

Pensando en estados
Niclas Nilsson

Dos fallos pueden hacer un acierto (y es difícil de arreglar)
Allan Kelly

Las herramientas Unix son tus amigas
Diomidis Spinellis

Los registros detallados perturbarán tu sueño
Johannes Brodwall

Cuando Programadores y Testers colaboran
Janet Gregory

Escribe pequeñas funciones usando ejemplos
Keith Braithwaite

Preocúpate por el código
Pete Goodliffe

CONSEJO 1 DE 97

Actúa con prudencia

Por Seb Rose · Traducción: Natán Calzolari

IDEA CLAVE

> "En todo lo que emprendas, actúa con prudencia y considera las consecuencias" ***Anónimo***

No importa qué tan cómoda se vea una agenda de trabajo al comienzo de una iteración, no podrás evitar sentirte bajo presión en algún momento. Si te encuentras en una situación en la que tienes que elegir entre "*hacerlo bien*" o "*hacerlo rápido*", suele ser tentador "hacerlo rápido" y pensar que regresarás a corregirlo más adelante. Cuando te haces esta promesa a ti mismo, a tu equipo, al cliente, lo haces en serio. Pero a menudo la siguiente iteración trae nuevos problemas y te debes enfocar en ellos. Este tipo de trabajo aplazado se conoce como **deuda técnica** y no es un buen amigo. Martin Fowler, en su taxonomía de la deuda técnica, la llama específicamente **deuda técnica** deliberada, la cual no debería confundirse con la deuda técnica inadvertida.

La deuda técnica es como un préstamo: te trae beneficios en el corto plazo, pero deberás pagar intereses hasta terminar de saldarla. Tomar atajos a la hora de programar hace que sea más difícil agregar funcionalidad o **refactorizar** tu código; las soluciones rápidas son un caldo de cultivo para defectos y casos de prueba muy frágiles. Mientras más tiempo las abandones, peor se ponen. Para cuando te decidas a corregir el problema puede que haya toda una pila de malas decisiones de diseño acumulada encima del problema original, haciendo que el código sea mucho más difícil de refactorizar y corregir. De hecho, es sólo cuando las cosas están tan mal como para tener que arreglarlas, que realmente vuelves y corriges el problema. Pero para entonces suele ser tan difícil corregirlo que no te puedes permitir el tiempo ni correr el riesgo.

Hay ocasiones en las que debes incurrir en la deuda técnica para cumplir con una fecha límite o para implementar una pequeña parte de una función. Intenta esquivar esos casos; sólo hazlo si la situación lo exige. Pero (y éste es un gran pero) debes mantener un ojo sobre la deuda técnica y pagarla tan pronto como puedas o las cosas se irán rápidamente cuesta abajo. Apenas te hayas endeudado, escribe una tarjeta o registra el problema en tu sistema de seguimiento para asegurarte de no olvidarlo.

Si planeas pagar la deuda en la próxima iteración, el costo será mínimo. Pero si la abandonas, se incrementarán los intereses y esto también deberá registrarse para que el costo permanezca a la vista. Hacer esto resaltará el impacto que tiene la deuda técnica del proyecto sobre el valor de la empresa y permitirá una priorización de pago. Cómo calcular y realizar el seguimiento de los intereses dependerá de cada proyecto, pero deberás hacerlo.

Paga la deuda técnica tan pronto como puedas; sería imprudente no hacerlo.

Aplica los principios de la programación funcional

Por Edward Garson · Traducción: Natán Calzolari

Recientemente, la comunidad programadora ha demostrado un renovado interés por la **programación funcional**. Parte del motivo es que las propiedades emergentes de este paradigma las hacen una buena opción para abordar la transición de la industria hacia el desarrollo sobre arquitecturas multi-core. Sin embargo, aunque es, sin duda, una aplicación importante, no es la razón por la que este texto te exhorta a que aprendas sobre **programación funcional**.

Dominar el paradigma funcional puede mejorar enormemente la calidad del código que escribes en otros contextos. Si lo comprendes y lo aplicas a tus diseños, lograrás un nivel mucho más alto de **transparencia referencial**.

La **transparencia referencial** es una cualidad deseable: implica que las funciones devuelvan siempre los mismos resultados cuando se les pase el mismo valor, independientemente de dónde y cuándo se las invoque. Es decir, la evaluación de una función no depende tanto de los efectos colaterales del estado mutable –idealmente, no depende en absoluto–.

Una de las principales causas de defectos cuando se programa en lenguajes imperativos no es otra que las variables mutables. Cualquier persona que se encuentre leyendo esto habrá tenido que investigar alguna vez por qué un valor no es el esperado en una situación particular. La semántica de visibilidad puede ayudar a mitigar estos errores insidiosos o, al menos, reducir drásticamente su ubicación; pero es probable que el verdadero culpable de su existencia sea un desarrollo que hace uso de mutabilidad excesiva.

Y la industria no nos ayuda mucho con este problema. La mayoría de la documentación introductoria sobre orientación a objetos tácitamente promueve este tipo de prácticas, porque a menudo utilizan como ejemplo una serie de objetos con un tiempo de vida relativamente largo, invocando métodos mutadores unos sobre otros, lo cual puede ser peligroso. Sin embargo, con un buen **desarrollo guiado por pruebas**, particularmente asegurándose de "simular roles, no objetos", se puede evitar la mutabilidad excesiva.

El resultado neto será un diseño que generalmente posee una mejor distribución de responsabilidades con una mayor cantidad de funciones –más pequeñas– que trabajan sobre los argumentos que se les pasa, en lugar de hacer referencia a miembros mutables. Habrá menos defectos y también será menos complejo detectarlos, porque es más fácil localizar dónde se introdujo un valor no deseado que deducir el contexto específico que resulta en una asignación errónea. Un diseño de este tipo establecerá un nivel mucho más alto de transparencia referencial; y, de seguro, nada fijará mejor estas ideas en tu cabeza que estudiar un lenguaje de programación funcional, en el cual este modelo de computación es la norma.

Por supuesto, este enfoque no es la mejor opción para todas las situaciones. Por ejemplo, en sistemas orientados a objetos de este estilo suele lograr mejores resultados con el desarrollo del **modelo de dominio** (es decir, en el cual la interacción de las funciones sirve para descomponer la complejidad de las reglas de negocio) y no tanto con el desarrollo de la interfaz de usuario.

Domina el paradigma de la programación funcional y podrás –con criterio– aplicar en otros contextos las lecciones que aprendas. Tus sistemas orientados a objetos (para empezar) se llevarán mejor con las bondades de la transparencia referencial y, contrario a lo que muchos te dirán, estarán más cerca de su contraparte funcional. De hecho, algunos incluso afirman que, en el fondo, los paradigmas de programación funcional y orientada a objetos no son más que un mero reflejo el uno del otro, una especie de yin y yang computacional.

CONSEJO 3 DE 97

Averigua qué haría el usuario (tú no eres el usuario)

Por Giles Colborne · Traducción: Natán Calzolari

Todos tendemos a asumir que los demás piensan como nosotros, pero no es así. Los psicólogos lo llaman efecto del falso consenso. Cuando la gente piensa o actúa de un modo diferente a nosotros es muy probable que (subconscientemente) los consideremos defectuosos en cierto modo.

Este prejuicio explica por qué a los programadores les cuesta tanto ponerse en el lugar de los usuarios. Los usuarios no piensan como programadores. Para empezar, pasan mucho menos tiempo usando computadoras y no saben, ni les interesa, cómo funcionan. Esto significa que no pueden recurrir a ninguna de las pilas de técnicas para resolver problemas que son tan comunes entre programadores. Los usuarios no saben reconocer los patrones ni indicaciones que los programadores manejan para trabajar y lidiar con las interfaces.

La mejor manera de entender cómo piensan los usuarios es observándolos. Pídele a un usuario que realice una tarea utilizando una aplicación similar a la que estás desarrollando. Asegúrate de que sea una tarea en serio: "agrega una columna de números" está bien; "calcula tus gastos del mes pasado" es mejor. Evita tareas muy específicas, como "¿puedes seleccionar estas celdas y agregar una fórmula SUMA debajo?"; es una pregunta algo obvia. Haz que el usuario te explique en detalle el proceso que realiza. No lo interrumpas. No intentes ayudarlo. Pregúntate todo el tiempo por qué está haciendo eso.

Lo primero que notarás es que los usuarios realizan una serie de cosas de manera similar. Intentan completar las tareas en el mismo orden y cometen los mismos errores en los mismos lugares. Deberías diseñar tu aplicación en torno a esta conducta base. Esto es algo que difiere de las reuniones de diseño, en las cuales se suelen hacer preguntas como: "¿y si el usuario quisiera…?". Estos planteamientos conducen al desarrollo de funciones demasiado complejas y generan confusión sobre lo que los usuarios realmente desean. Observarlos eliminará esta confusión.

Verás que los usuarios suelen atascarse. Cuando tú te atascas, buscas una solución. Cuando los usuarios se atascan, reducen su foco de atención; se les vuelve más complicado ver una solución al problema en otro lugar de la pantalla. Ésta es una de las razones por las que los textos de ayuda son una mala solución al mal diseño de interfaces de usuario. Si debes agregar instrucciones o textos de ayuda, asegúrate de hacerlo justo al lado de las áreas problemáticas. Esta limitación de los usuarios es el motivo por el que los tooltips son más útiles que los menús de ayuda.

Los usuarios tienden a salir del paso de alguna manera. Encontrarán algo que funcione y se aferrarán a ello sin importar lo complejo que sea, pero es mejor proveer un modo obvio de hacer las cosas que dos o tres atajos.

También te encontrarás con que hay una marcada diferencia entre lo que los usuarios dicen que quieren y lo que realmente quieren. Lo cual es preocupante, ya que para averiguar los requerimientos lo normal es preguntarles. Es por esto que el mejor modo de relevar los requerimientos es observando a los usuarios. Pasar una hora con ellos es mucho más informativo que pasar un día suponiendo qué quieren.

Automatiza el estándar de codificación

Por Filip van Laenen · Traducción: Natán Calzolari

Probablemente a ti también te sucedió. Al comenzar un proyecto todo el mundo tiene buenas intenciones; las llamaremos "resoluciones de proyecto nuevo". A menudo, muchas de estas resoluciones se documentan, y las que tienen que ver con el código terminan en el **estándar de codificación** del proyecto. Durante la primera reunión, el jefe de desarrollo revisa la documentación y, en el mejor de los casos, todos aceptan que intentarán respetarla. Sin embargo, una vez que el proyecto se pone en marcha, las buenas intenciones se van dejando de lado, una a una. Para cuando se entrega el proyecto, el código es un desastre y nadie parece saber por qué.

¿En qué momento salieron mal las cosas? Probablemente desde la reunión inicial. Algunos miembros no estaban prestando atención; otros no lo consideraron importante. Para peor, algunos no estuvieron de acuerdo y ya estaban planeando rebelarse en contra del estándar. Por último, algunos sí lo comprendieron y estuvieron de acuerdo pero, cuando la presión del proyecto fue demasiada, tuvieron que dejar de lado algunas convenciones. Aplicar un buen formato al código no te hará ganar puntos con un cliente que desea más funcionalidad. De hecho, respetar un **estándar de codificación** puede ser bastante aburrido si la función no está automatizada: intenta indentar una clase a mano para comprobarlo por tu cuenta.

Pero si es tan problemático, ¿para qué queremos un estándar de codificación? Una de las razones para darle un formato uniforme al código es que, de este modo, nadie se "adueñará" del código que escriba utilizando un formato propio. Probablemente queremos evitar que los programadores utilicen ciertos antipatrones, para así ahorrarnos algunos errores comunes. En general, un estándar de codificación debería hacer más fácil el trabajo grupal de un proyecto y mantener la velocidad de desarrollo desde el principio hasta el final. Se deduce entonces que todos deberían estar de acuerdo con el estándar; no ayuda que un programador utilice tres espacios para indentar y otro utilice cuatro.

Hay una gran cantidad de herramientas que se pueden usar para producir reportes de calidad de código, y para documentar y mantener el estándar de codificación, pero ésa no es la solución completa. El estándar debería automatizarse e imponerse siempre que sea posible. Por ejemplo, de las siguientes maneras:

Asegúrate de que parte del proceso de compilación sea darle formato al código, de modo que todo el mundo lo realice cada vez que se compile la aplicación.

Utiliza herramientas de análisis de código estático para encontrar antipatrones. Si se encuentra alguno, detén la compilación.

- Aprende a configurar estas herramientas para que detecten antipatrones definidos por ti mismo y para tus proyectos específicos.
- Mide la **cobertura del código**, pero también evalúa automáticamente los resultados. Nuevamente, detén la compilación si los resultados son muy bajos.

Intenta aplicar esto en todo lo que consideres de importancia, aunque no te será posible automatizarlo todo. Las cosas que no puedas marcar o corregir automáticamente podrían agruparse en un conjunto de directrices suplementarias al estándar automatizado, pero ten en cuenta que probablemente tú y tus colegas no lo respeten con la misma diligencia.

Por último, el estándar de codificación debería ser dinámico y no estático. A medida que el proyecto evolucione, sus necesidades también irán cambiando, y lo que quizás pareció inteligente en un principio, no será necesariamente inteligente algunos meses después.

La belleza está en la simplicidad

Por Jørn Ølmheim · Traducción: Natán Calzolari

Hay una gran cita de Platón que es particularmente importante que los programadores sepamos y recordemos siempre: “La belleza en el estilo, la armonía, la gracia y el buen ritmo dependen de la **simplicidad**”. Creo que esta cita resume en una sola oración todos los valores a los que deberíamos aspirar los desarrolladores de software.

En nuestro código, nos esforzamos por lograr una serie de cosas:

- **Legibilidad**
- **Mantenibilidad**
- Velocidad de desarrollo
- La esquiva cualidad de la belleza

Platón nos está diciendo que el factor que nos permitirá alcanzar todas estas cualidades es la **simplicidad**.

¿Pero qué hace bello al código? Ésta puede ser una pregunta muy subjetiva. La percepción de la belleza depende mucho de nuestro trasfondo individual, tal como sucede con cualquier otra cosa. La gente formada en las artes tiene una percepción (o enfoque) sobre la belleza que es distinta a la de la gente formada en las ciencias. En el ámbito del arte se tiende a analizar la belleza del software comparándola con obras de arte, mientras que en el de las ciencias se habla de la simetría y la proporción áurea; se intenta reducir las cosas a fórmulas. En mi experiencia, la simplicidad es la base de los argumentos en ambos lados de la moneda.

Piensa en el código que has estudiado. Si no has pasado un buen tiempo leyendo el código de alguien más, deja de leer esto ahora mismo y ve a buscar algo de software libre para estudiar. ¡En serio, no es broma! Busca en Internet algo de código en tu lenguaje preferido, escrito por algún experto reconocido.

¿Ya has regresado? Bien. ¿Dónde estábamos? Ah, sí… Me he encontrado con que el código que me llama la atención y que considero hermoso siempre posee una misma serie de características. La más importante es la simplicidad. Me encuentro con que, sin importar qué tan complicada sea la aplicación o sistema en su totalidad, las partes individuales deben mantenerse simples: los objetos deben ser sencillos, poseer una única responsabilidad y contener métodos similarmente simples, con una tarea bien definida y nombres descriptivos. Algunos piensan que la idea de escribir métodos breves, de entre cinco y diez líneas de código cada uno, es bastante extrema, y algunos lenguajes hacen que sea muy difícil lograr esto, pero yo creo que esta brevedad es un objetivo deseable.

En resumen, para que el código sea bello debe ser simple. Cada pieza individual debe ser sencilla, y poseer responsabilidades y relaciones simples con otras partes del sistema. De este modo se logra que nuestros

proyectos puedan mantenerse en el tiempo, con **código limpio**, sencillo y verificable, lo cual permite mantener una alta velocidad de desarrollo durante el tiempo de vida del proyecto.

La belleza nace y se encuentra en la simplicidad.

CONSEJO 6 DE 97

Antes de Refactorizar

Por Rajith Attapattu · Traducción: Espartaco Palma

En algún punto todo programador necesitará **refactorizar** código existente. Pero antes de hacerlo por favor piensa en lo siguiente, ya que tú y otras personas podrían ahorrar una gran cantidad de tiempo (y dolor):

> El mejor enfoque para la reestructuración comienza por hacer un balance del **código base** existente y las pruebas escritas contra ese código. Esto ayudará a entender las fortalezas y debilidades del código en su estado actual, por lo que puedes asegurar que retienes los puntos fuertes, mientras evitas los errores. Todos pensamos que podemos hacerlo mejor que el sistema actual... hasta que terminamos con algo que no es mejor –o incluso peor– que la anterior encarnación, debido a que fallamos en aprender de los errores existentes en el sistema.
>
> Evita la tentación de volver a escribir todo. Es mejor reusar tanto código como sea posible. No importa que tan feo sea el código, ya ha sido probado, revisado, etcétera. Desechar el código viejo –especialmente si está en producción– significa que estás desechando meses (o años) de pruebas sobre el aguerrido código que podría haber tenido ciertos atajos y correcciones críticas de errores de los cuales no estás enterado. Si no tomas esto en cuenta, el nuevo código que se escriba podría terminar mostrando el mismo error misterioso que fue reparado en el código antiguo. Esto desperdiciará un montón de tiempo, esfuerzo y conocimiento adquiridos a través de los años.
>
> Muchos cambios incrementales son mejores que un cambio masivo. Los cambios incrementales permiten medir el impacto en el sistema más fácilmente a través de la retroalimentación, como las pruebas. No es divertido ver cientos de pruebas fallidas después de realizar un cambio. Esto puede conducir a la frustración y presión que puede, a su vez, dar lugar a malas decisiones. Un par de pruebas fallidas es fácil de manejar y provee un enfoque más manejable.
>
> Después de cada iteración es importante asegurar que las pruebas existentes pasan. Agrega nuevas pruebas si las pruebas existentes no son suficientes para cubrir los cambios realizados. No deseches las pruebas del código antiguo sin la debida consideración. En la superficie algunas de estas pruebas podrían no parecer aplicables a tu nuevo diseño, pero será de utilidad el esfuerzo de investigación a fondo de las razones por las cuales estas pruebas en particular fueron añadidas.
>
> Las preferencias personales y el ego no deben ponerse en el camino. Si algo no está roto, ¿para qué arreglarlo? Que el estilo o la estructura del código no se ajuste a tus preferencias personales no es una razón válida para reestructurarlo. Pesar que podrías hacer un mejor trabajo que el programador previo no es una razón válida tampoco.
>
> La nueva tecnología es razón insuficiente para refactorizar. Una de las peores razones para refactorizar se debe a que el código actual está muy por detrás de las buenas tecnologías que tenemos hoy en día, y creemos que un nuevo lenguaje o framework puede hacer las cosas mucho más elegantemente. A menos que un análisis de costo-beneficio muestre que el nuevo lenguaje o framework beneficiará la funcionalidad, mantenimiento o productividad, es mejor dejar las cosas como están.

Recuerda que los humanos cometen errores. Reestructurar no siempre garantiza que el nuevo código será mejor o tan bueno como el intento anterior. He visto y sido parte de muchos intentos de reestructuración fallidos. No fue bonito, pero fue humano.

Ten cuidado al compartir

Por Udi Dahan · Traducción: Espartaco Palma

Era mi primer proyecto en la compañía. Había terminado mi carrera y estaba ansioso por probarme a mí mismo, me quedaba tarde cada día a revisar el código existente. Conforme trabajaba en mi primera característica tomaba cuidados adicionales para poner en marcha cada cosa que había aprendido: comentarios, bitácoras, sacando código compartido a bibliotecas de ser posible, el trabajo. La revisión de código de la que había sentido tan listo vino como una sorpresa desagradable: ¡el reúso estaba mal visto! ¿Cómo podía ser eso posible? En toda la universidad el reúso era tomado como el epítome de la ingeniería de calidad de software. Todos los artículos que había leído, los libros de textos, lo que me habían enseñado los profesionales de software con experiencia. ¿Estaba todo mal?

Resulta que había olvidado algo crítico.

Contexto.

El hecho de que dos partes muy diferentes del sistema realizaran la misma lógica de la misma manera significaba menos de lo que pensaba. Hasta que saqué esas bibliotecas de código compartido, esas partes no eran dependientes una de otra. Cada una podían evolucionar independientemente. Cada una podía cambiar su lógica para satisfacer las necesidades de los cambios en el entorno empresarial del sistema. Esas cuatro líneas de código similar fueron accidentales, una anomalía temporal, una coincidencia. Es decir, hasta que llegué.

Las bibliotecas de código compartido que había creado ataban los cordones de cada zapato de cada pie entre ellos. Los pasos por un dominio de negocio no podrían ser hechos sin primero sincronizarlos. Los costos de mantenimiento en estas funciones independientes solían ser insignificantes, pero la biblioteca común requería un orden de magnitud de más pruebas.

A pesar de que había disminuido el número absoluto de líneas de código en el sistema, había incrementado el número de dependencias. El contexto de esas dependencias es crítico, si hubieran sido localizadas, podían haber sido justificadas y tendrían algún valor positivo. Cuando estas dependencias no se mantienen bajo control, sus tentáculos se enredan en las más grandes preocupaciones del sistema, a pesar de que el código en sí se ve muy bien.

Estos errores son insidiosos por eso, en esencia, suenan como una buena idea. Cuando se aplican en el contexto adecuado, estas técnicas son valiosas. En el contexto equivocado, incrementan el costo en vez del

valor. Hoy en día soy mucho más cuidadoso en los temas de compartir cuando entro en un **código base** existente sin el conocimiento del contexto en el que se utilizan las distintas partes.

Cuidado al compartir. Revisa tu contexto. Sólo entonces, procede.

CONSEJO 8 DE 97

La Regla Boy Scout

Por Uncle Bob · Traducción: Espartaco Palma

Los Boy Scout tienen una regla: “Siempre deja el lugar de acampamento más limpio que como lo encontraste”. Si encuentras un desastre en el piso, lo limpias sin importar quién pudo haber hecho el desastre. Mejoras intencionalmente el ambiente para el siguiente grupo de campistas. En realidad, la forma original de la regla, escrita por Robert Stephenson Smyth Baden-Powell, el padre del Scoutismo, era “Intenta y deja el mundo un poco mejor que como lo encontraste”.

Que tal si seguimos una regla similar con nuestro código: “siempre deja un módulo más limpio que cuando lo revisaste”. No importa quién fue el autor original, qué tal si siempre hacemos algún esfuerzo, sin importar lo pequeño, para mejorar el módulo. ¿Cuál sería el resultado?

Creo que si todos seguimos esa simple regla, podría ser el final del implacable deterioro en nuestros sistemas. En vez de ello, nuestros sistemas serían gradualmente mejores y mejores en cuanto evolucionaran. También veríamos equipos que cuidan el sistema como un todo, en vez de individualistas cuidando su pequeña partecita.

No creo que esta regla sea mucho pedir. No tienes que hacer cada módulo perfecto antes de dejarlo. Simplemente tienes que hacerlo un poco mejor cuando lo dejes. Claro, esto significa que cualquier código que agregues al módulo debe estar limpio. Esto también significa que limpies, al menos, alguna otra cosa antes de que regreses el módulo. Podrías simplemente mejorar el nombre de una variable o separar una larga función en dos pequeñas. Podrías romper una dependencia circular o agregar una interfaz para desacoplar la política del detalle.

Francamente, esto me suena como decencia común, como lavarte las manos después de usar el baño o poner la basura en el bote en vez de tirarla en el suelo. De hecho, el acto de dejar un desastre en el código debería ser socialmente inaceptable, como tirar basura. Esto debería ser algo que simplemente no se hace.

Pero es más que eso. El cuidado de nuestro propio código es una cosa. Tener cuidado del código del equipo es otra muy distinta. Los equipos se ayudan entre sí, y después se limpian entre ellos. Siguen la regla Boy Scout, porque es bueno para todos, no sólo para ellos.

CONSEJO 9 DE 97

Primero revisa tu código antes de buscar culpar a otros

Por Allan Kelly · Traducción: Espartaco Palma

Los desarrolladores –¡todos nosotros!– frecuentemente tenemos problemas creyendo que nuestro propio código está roto. Es tan improbable que, por una sola vez, debe ser el compilador el que no funciona.

Aunque la verdad es muy (muy) inusual que el código no funcione debido a un bug en el compilador, intérprete, sistema operativo (SO), servidor de aplicaciones, base de datos, gestor de memoria o cualquier otra parte del software del sistema. Sí, esos bugs existen, pero son mucho menos comunes de lo que quisieras creer.

Una vez tuve un genuino problema con un error en el compilador al optimizar un ciclo variable, pero he imaginado que mi compilador o SO ha tenido un bug muchas más veces. He desperdiciado un montón de tiempo, horas de soporte y tiempo de gestión en el proceso sólo para sentirme un poco tonto cada vez que resultó ser mi error después de todo.

Asumiendo que las herramientas son ampliamente usadas, maduras y empleadas en varias pilas de tecnología, hay muy pocas razones para dudar de la calidad. Por supuesto, si la herramienta es un "***early release***", o usada sólo por una pocas personas en todo el mundo, o una pieza raramente usada, versión 0.1, Software Libre (***Open Source***) puede haber buenas razones para sospechar del software (igualmente, una versión alfa de un software comercial podría ser sospechosa).

Teniendo en cuenta qué tan raros son los errores del compilador, estás mucho mejor poniendo tu tiempo y energía en encontrar el error en tu código que probando que el compilador está mal. Todos los consejos comunes en la depuración aplican, así que aísla el problema, apaga las llamadas, rodéalo con pruebas; revisa convenciones de llamada, bibliotecas compartidas y números de versión; explícalo a alguien más; busca corrupciones de pilas y tipos de variables que no coinciden; prueba el código en diferentes máquinas y con diferentes configuraciones de compilación, como el debug y la liberación.

Cuestiona tu propias suposiciones y las suposiciones de otros. Las herramientas de diferentes proveedores pueden tener diferentes suposiciones dentro de ellas, así también podrían diferir las herramientas del mismo proveedor.

Cuando alguien más está reportando un problema que no puedes duplicar, ve y mira qué está haciendo. Ellos podrían estar haciendo algo que nunca pensaste o están haciendo algo en diferente orden.

Como una regla personal, si tengo un error que no puedo precisar, y empiezo a pensar que es el compilador, entonces es tiempo de mirar daños en la pila. Esto es especialmente cierto si la adición de rastreo de código hace que el problema se vaya..

Los problemas multihilo son otra fuente de errores que convierte el cabello en gris e induce gritarle a la máquina. Todas las recomendaciones para favorecer el código simple se multiplican cuando un sistema es multihilo. No se puede confiar en la revisión de errores y las pruebas unitarias para encontrar tales errores con cierta coherencia, así que la **simplicidad** de diseño es fundamental.

Así que antes de apresurarte en culpar al compilador, recuerda el consejo de Sherlock Holmes: “Una vez que elimines lo imposible, lo que quede, sin importar que tan improbable parezca, debe ser verdad”, aunque yo prefiero el consejo de Drik Gently: “Una vez que eliminas lo improbable, lo que quede, sin importar que tan imposible sea, debe ser verdad”.

Escoge tus herramientas con cuidado

Por Giovanni Asproni · Traducción: Espartaco Palma

Las aplicaciones modernas rara vez son construidas desde cero. Se ensamblan usando herramientas existentes –componentes, bibliotecas y frameworks– por una serie de buenas razones:

> Las aplicaciones crecen en tamaño, complejidad y sofisticación, mientras el tiempo para desarrollarlas decrece. Se hace un mejor uso del tiempo e inteligencia del desarrollador, si pueden concentrarse en escribir más código del dominio del negocio y menos código de infraestructura
>
> Los componentes y frameworks ampliamente utilizados con frecuencia tienen menos errores que aquellos desarrollados en casa.
>
> Hay un montón de software de alta calidad disponible en la red de forma gratuita, lo cual significa menores costos de desarrollo y mayor probabilidad de encontrar desarrolladores con el interés y experiencia necesaria.
>
> La producción y mantenimiento de software es un trabajo humanamente intensivo, por lo que comprarlo podría ser más barato que construirlo.

Sin embargo, escoger la mezcla completa de herramientas para tu aplicación puede ser un negocio riesgoso que requiere pensarlo un poco. De hecho, hay unas cuantas cosas que deberías tener en mente mientras estás haciendo la elección:

> Las diferentes herramientas pueden estar basadas en distintos supuestos sobre su contexto –por ejemplo, la infraestructura circundante, modelo de control, modelo de datos, protocolos de comunicación, etcétera – lo cual puede llevar a un diferencial de arquitectura entre la aplicación y las herramientas. Dichas diferencias conducen a ***hacks*** y ***workarounds*** que harán el código más complejo de lo necesario.
>
> Las diferentes herramientas tienen diferentes ciclos de vida, y actualizar una de ellas podría convertirse en algo extremadamente difícil y una tarea que consume tiempo en cada nueva funcionalidad, cambios de diseño o incluso correcciones de errores que podrían causar incompatibilidades con las otras herramientas. Entre más grande sea el número de herramientas, peor es el problema en el que puede convertirse.
>
> Algunas herramientas requieren configuraciones, lo que frecuentemente significa uno o más archivos `XML`, lo cual se sale de control muy rápido. La aplicación puede terminar como si fuese escrita toda en XML más unas cuántas líneas de código en algún lenguaje de programación. La complejidad en la configuración hará la aplicación difícil de mantener y de extender.
>
> Ocurre un vendor-lock cuando el código que depende en gran medida en un proveedor específico termina siendo arriesgado por él en varias formas: mantenimiento, rendimiento, habilidad para evolucionar, precio, etc.

> Si planeas usar software libre, puedes descubrir que no es tan libre después de todo. Quizás necesites comprar soporte comercial, lo cual no necesariamente va a ser barato.
>
> Los términos de licenciamiento importan, incluso para el software libre. Por ejemplo, en algunas compañías no es aceptable usar software licenciado bajo los términos de la licencia GNU, debido a su naturaleza viral, es decir, el software desarrollado con él debe ser distribuido junto con su código fuente.

Mi estrategia personal para mitigar estos problemas es comenzar poco a poco, usando sólo las herramientas que son absolutamente necesarias. Usualmente el enfoque inicial está en quitar la necesidad de participar en la programación (y los problemas) de infraestructura de bajo nivel, por ejemplo, usando algún middleware en vez de usar sockets para aplicaciones distribuidas. Y entonces agregar más si es necesario. También tiendo a aislar las herramientas externas de mis objetos de dominio del negocio con respecto a interfaces y capas de presentación, así puedo cambiar la herramienta, si lo tengo que hacer, con sólo una pequeña dosis de dolor. Un lado positivo de este enfoque es que generalmente termino con una aplicación más pequeña que usa menos herramientas externas de lo que originalmente se pronosticó.

Codifica en el lenguaje del dominio

Por Dan North · Traducción: Espartaco Palma

Imagínate dos códigos bases. En uno te encuentras esto:

EJEMPLO DE CÓDIGO

```
if (portfolioIdsByTraderId.get(trader.getId())
  .containsKey(portfolio.getId())) {...}
```

Te rascas la cabeza imaginándote para que podría servir este código. Parece que está obteniendo un ID desde un objeto comerciante ("trader"), usándolo para obtener aparentemente un mapa de mapas y, entonces, está viendo si otro ID desde un objeto portafolio ("portfolio") existe en el mapa interior. Te rascas la cabeza un poco más. Ves la declaración del método `portfolioIdsByTraderId` y descubres esto:

EJEMPLO DE CÓDIGO

```
Map<int, Map<int, int>> portfolioIdsByTraderId;
```

Poco a poco te das cuenta que podría tener algo que ver con que un comerciante tenga acceso a un portafolio en particular. Y, por supuesto, encontrarás el mismo fragmento de búsqueda –o un similar-pero-ligeramente-diferente fragmento de código– en el momento en que a alguien le importa si un comerciante tiene acceso a un portafolio en particular.

En el otro **código base** te encuentras con esto:

EJEMPLO DE CÓDIGO

```
if (trader.canView(portfolio)) {...}
```

No hay rascado de cabeza. No necesitas saber cómo lo sabe un comerciante. Quizás es uno de esos mapas de mapas escondidos dentro. Pero es un asunto del comerciante, no tuyo.

Ahora, ¿en cuál de estos códigos te gustaría estar trabajando?

Hubo un tiempo en que sólo teníamos unas muy básicas estructuras de datos: bit, bytes y caracteres (realmente sólo bytes que pretendíamos que fueran letras y puntuaciones). Tener decimales eran un poco truculento porque nuestros números de base 10 no trabajan muy bien en binario, así que teníamos varios tamaños de tipos de punto flotante. Entonces vinieron las matrices y las cadenas (realmente sólo matrices distintas). Teníamos pilas, colas, hashes, listas ligadas y listas salteadas y muchas otras excitantes

estructuras de datos que no existían en el mundo real. La "Ciencia Computacional" se trataba de gastar mucho esfuerzo mapeando el mundo real en nuestras estructuras de datos restrictivas. Los verdaderos gurús podrían incluso recordar cómo lo habían logrado.

¡Entonces tuvimos los tipos definidos por el usuario! Está bien, esto no es noticia, pero fue un cambio en el juego, de alguna manera. Si tu dominio contiene conceptos como negociantes y portafolios, podías modelarlos con tipos llamados, digamos, Comerciantes y Portafolio. Pero, más importante que esto, también puedes modelar relaciones entre ellos usando términos de dominio.

Si no codificas usando términos del dominio estás creando un entendimiento tácito (léase: secreto) de que este valor de tipo entero que está por ahí significa la manera de identificar a un comerciante, donde ese valor de tipo entero por allá es la manera de identificar un portafolio. (¡Mejor no confundirlos!) Y si representas un concepto de negocio ("a algunos comerciantes no les está permitido ver algunos portafolios –es ilegal–") con un algoritmo, digamos la existencia de relaciones en un mapa de claves, no le estás haciendo ningún favor a los chicos de auditoría y quejas.

El programador de junto quizás no sepa el secreto, así que ¿porqué no hacerlo explícito? Usar una llave como el término de búsqueda de otra llave que realiza la revisión de una llave existente no es terriblemente obvio. ¿Cómo se supone que alguien intuya que ahí están implementadas las reglas de negocio que previenen conflictos de interés?

Realizar conceptos explícitos del dominio en tu código significa que otros programadores pueden adquirir la intención del código mucho más fácilmente que intentar meter un algoritmo en lo que entienden sobre el dominio. Esto también significa que cuando el modelo del dominio evoluciona –es decir, que tu entendimiento se incrementa– estés en una buena posición para evolucionar el código. En conjunto con una buena encapsulación, aumenta la oportunidad de que la regla exista sólo en un lugar y que puedes cambiarla sin que el código dependiente se dé cuenta.

El programador que venga unos cuantos meses después a trabajar con el código te lo agradecerá y quizás ese programador seas tú.

CONSEJO 12 DE 97

El código es diseño

Por Ryan Brush · Traducción: Espartaco Palma

Imagínate despertar mañana y saber que la industria de la construcción ha hecho el avance del siglo. Millones de robots baratos e increíblemente rápidos pueden fabricar materiales de la nada, tener gasto energético cercano a cero y se pueden reparar a sí mismos. Y se pone mejor: al darle un no-ambiguo plano para un proyecto de construcción, el robot puede construirlo sin la intervención humana, todo ello a un costo insignificante.

Uno puede imaginar el impacto en la industria de la construcción, pero ¿qué pasaría más adelante? ¿Cómo cambiaría el comportamiento de los arquitectos y diseñadores si los costos de construcción fueran insignificantes? Hoy en día modelos físicos y computacionales son creados y rigorosamente probados antes de invertir en la construcción. ¿Nos preocuparíamos si la construcción fuera esencialmente gratis? Si un diseño se colapsa, no hay problema, sólo encuentra qué estuvo mal y pon a nuestros robots mágicos a construir otro. Hay otras implicaciones. Con modelos obsoletos, los diseños sin terminar evolucionan mediante la construcción y mejoran en repetidas ocasiones hacia una aproximación de la meta final. Un observador casual podría tener problemas distinguiendo un diseño inacabado y un producto terminado.

Nuestra capacidad para predecir líneas de tiempo se esfumaría. Los costos de construcción son calculados más fácilmente que los costos de diseño –sabemos el costo aproximado de instalar una viga y cuántas vigas necesitamos–. Como las tareas predecibles se reducen a cero, la época del diseño menos predecible empieza a dominar. Los resultados se producen con mayor rapidez, pero los plazos fiables escapan.

Por supuesto, se sigue aplicando la presión de una economía competitiva. Con los costos de construcción eliminados, una empresa puede completar rápidamente un diseño ganando una esquina en el mercado. El tener pronto los diseños terminados se convierte en el empuje central de las firmas de ingeniería. Inevitablemente, alguien no familiarizado con el diseño verá una versión invalidada, ve una ventaja del mercado al liberar temprano y dice "esto parece lo suficientemente bien".

Algunos proyectos de vida o muerte serán más diligentes, pero en muchos casos los consumidores aprende a sufrir el diseño incompleto. Las empresas siempre puede mandar robots mágicos a "parchar" los edificios y vehículos rotos que venden. Todo esto apunta a una conclusión intuitiva: nuestra única premisa era una dramática reducción en los costos de construcción, con el resultado de que la calidad ha empeorado.

No debería sorprendernos que la historia de arriba fuera ejecutada por el software. Si aceptamos que el código es diseño –un proceso creativo en vez de uno mecánico– la crisis del software se explica. Ahora tenemos una crisis de diseño: la demanda de diseños validados y de calidad excede nuestra capacidad de crearlos. La presión por usar diseños incompletos es fuerte.

Afortunadamente, este modelo también ofrece pistas de cómo mejorar. Las simulaciones físicas equivalen a **pruebas automatizadas**; el diseño de software no está completo hasta que es validado con una batería de pruebas brutal. Para hacer tales pruebas más efectivas estamos encontrando maneras de frenar en el gran

espacio de estados de los grandes sistemas. Los lenguajes mejorados y las prácticas de diseño nos dan esperanza. Finalmente, hay un hecho ineludible: los grandes diseños son producidos por grandes diseñadores dedicados a la maestría de su oficio. El código no es diferente.

CONSEJO 13 DE 97

El diseño del código sí importa

Por Steve Freeman · Traducción: Espartaco Palma

Hace muchos años trabajaba en un sistema en Cobol, en el cual no se le permitía al personal cambiar la sangría a menos que tuvieran una razón para cambiar el código, debido a que alguien, alguna vez, descompuso algo al dejar un trozo de línea en una de las columnas especiales al inicio de una línea. Esto aplicaba incluso si el diseño estaba equivocado, lo cual sucedía algunas veces, así que teníamos que leer el código muy cuidadosamente porque no podíamos confiar en él. La política debió costar una fortuna en fricción de programador.

Hay una investigación que muestra que todos pasamos más de nuestro tiempo de programación navegando y leyendo código –encontrando dónde hacer el cambio– que escribiendo, así que esto es lo que queremos optimizar.

> Fácil de escanear. La gente es buena en la comparación de patrones visuales (una reminiscencia de la época en la que teníamos que observar leones en la sabana), así que puedo ayudarme al hacer todo lo que no es directamente relevante al dominio, toda la “complejidad accidental” que viene con muchos lenguajes comerciales, ocultarlo en el fondo de pantalla para estandarizarlo. Si el código que se comporta igual luce igual, entonces mi sistema perceptual me ayudará a escoger las diferencias. Es por eso que también observo las convenciones sobre cómo diseñar las partes de una clase dentro de una unidad de compilación: constantes, campos, métodos públicos, métodos privados.
>
> Diseño expresivo. Todos hemos aprendido que toma su tiempo encontrar los nombres adecuados para que nuestro código exprese, tan claramente como es posible, lo que hace; en lugar de sólo listar los pasos, ¿está bien? El diseño de código también es parte de esta expresividad. Un primer corte es tener el acuerdo del equipo en un formateo automático para lo básico, entonces podemos hacer ajustes manuales mientras estamos codificando. A menos que haya un disensión activa, el equipo convergerá rápidamente en un estilo de “acabado manual” común. Un formateador no puede entender mis intenciones (debería saberlo, una vez codifiqué uno), y es más importante para mí que los saltos de línea y los agrupadores reflejen la intención de mi código, no sólo la sintaxis del lenguaje (Ken McGuire me liberó de mi esclavitud a los formateadores automáticos de código).
>
> Formato compacto. Mientras más puedo conseguir en una pantalla, más puedo ver si se rompe el contexto al desplazarme o al cambiar de archivo, lo que significa que puedo dejar menos estados en mi cabeza. Los comentarios del procedimiento largos y los espacios en blanco tienen sentido para nombres de 8 caracteres e impresoras, pero ahora vivo en un `IDE` que hace el coloreo de sintaxis y el enlace cruzado. Los pixeles son mi factor limitante, así que quiero que cada uno contribuya hacia mi entendimiento del código. Quiero que el diseño me ayude a entender el código, pero no más que eso.

Un amigo no programador comentó alguna vez que el código se parece a la poesía. Tengo esa sensación en el código bueno, que todo en el texto tiene un propósito y está ahí para ayudarme a entender la idea. Desafortunadamente, escribir código no tiene la misma imagen romántica que escribir poesía.

CONSEJO 14 DE 97

Revisiones de código

Por Mattias Karlsson · Traducción: Espartaco Palma

Deberías hacer revisiones de código. ¿Por qué? Porque incrementan la calidad del código y reducen la tasa de defectos. Pero no necesariamente por las razones que podrías pensar.

Debido a que pudieron haber tenido algunas malas experiencias con las revisiones, muchos programadores tienden a rechazar las revisiones de código. He visto organizaciones que requieren que todo el código pase por una revisión formal antes de enviar a producción. Con frecuencia es el arquitecto o el líder de desarrollo quien hace esta revisión, una práctica que puede ser descrita como arquitecto revisando todo. Está escrito en el manual del proceso de desarrollo de software, así que los programadores deben acatar. Puede ser que algunas organizaciones necesiten tal rigidez y procesos formales, pero muchas no. En la mayoría de las organizaciones ese enfoque no es productivo. Los revisados pueden sentirse como que están siendo juzgados por una junta de libertad condicional. Los revisores necesitan tanto el tiempo para leer el código como el tiempo para estar al día con todos los detalles del sistema. Los revisores se pueden convertir rápidamente en cuellos de botella en este proceso, y el proceso se degenera pronto.

En vez de limitarse a corregir errores en el código, el propósito de las revisiones de código debería ser compartir conocimiento y establecer guías comunes de codificación. Compartir tu código con otros programadores habilita la propiedad colectiva de código. No limites su flujo, deja que cualquier miembro del equipo revise el código con el resto del equipo. En vez de buscar errores deberías revisar el código tratando de aprenderlo y entenderlo.

Sé gentil durante las revisiones de código. Asegúrate de que los comentarios sean constructivos, no cáusticos. Introduce diferentes roles en la junta de revisión, evitar tener a los más viejos dentro del equipo afecta las revisiones de código. Los ejemplos de roles pueden incluir algún revisor enfocado en la documentación, otro en excepciones y un tercero en busca de funcionalidad. Este enfoque ayuda a distribuir la carga de las revisiones a través de los miembros del equipo.

Haz la revisión de código con regularidad, un día a la semana. Usa un par de horas en la junta de revisión. Rota a los que tuvieron revisión semanalmente en un patrón simple de ***round-robin***. Recuerda también cambiar roles entre los miembros del equipo en cada junta de revisión. Involucra a los novatos en las revisiones de código. Puede que sean inexpertos, pero su conocimiento universitario reciente puede proveer una perspectiva diferente. Involucra expertos por su experiencia y conocimiento; identificarán código propenso a errores más rápido y con mayor precisión. Las revisiones de código fluirán más fácilmente si el equipo tiene convenciones de codificación que se comprueban mediante herramientas. De este modo, el formato del código nunca será discutido durante la junta de revisión de código.

Hacer las revisiones de código divertidas es quizás el factor más importante para el éxito. Las revisiones se tratan de la gente en revisión. Si la junta de revisión es dolorosa o aburrida será más difícil motivar a cualquiera. Que sea una revisión de código informal, cuyo propósito principal sea compartir conocimiento entre los miembros del equipo. Deja los comentarios sarcásticos fuera y trae un pastel o almuerzo en bolsa café en su lugar.

Codificando con la razón

Por Yechiel Kimchi · Traducción: Espartaco Palma

Trata de averiguar manualmente la correctitud de software resulta en una prueba formal más larga y propensa a errores que el código mismo. Las herramientas automatizadas son preferibles, pero no siempre posibles. Lo siguiente describe una ruta intermedia: razonamiento semi-formal sobre la dicha correctitud.

El planteamiento de fondo es dividir todo el código en cuestión de secciones cortas –desde una sola línea, como invocar a una función, hasta bloques de menos de 10 líneas– y discutir acerca de su exactitud. Los argumentos sólo necesitan ser suficientemente fuertes para convencer al compañero del diablo como tu pareja de programación.

Una sección debería ser elegida de modo que en cada terminal el estado del programa (léase: el conteo del programa y los valores de todos los objetos “vivos”) satisface una propiedad fácilmente descrita y que la funcionalidad de esa sección (transformación de estado) sea fácil de describir como una sola tarea –estos harán el razonamiento más sencillo–. Tales propiedades terminales generalizan conceptos como precondinción y poscondición de funciones, e invariantes para ciclos y clases (con respecto a sus instancias). La lucha para que las secciones sean independientes de las otras tanto como sea posible simplifica el razonamiento y es indispensable cuando estas secciones son modificadas.

Muchas de las prácticas de codificación que son bien conocidas (aunque quizás menos seguidas) y consideradas “buenas” hacen el razonamiento más fácil. Por lo tanto, sólo con la intención de razonar sobre tu código ya estás comenzando a pensar acerca de un mejor estilo y estructura. Como era de esperarse, la mayoría de estas prácticas pueden ser revisadas por analizadores de código estático:

- Evita usar sentencias `goto`, ya que hacen las secciones remotas altamente interdependientes
- Evita usar variables globales modificables, debido a que hacen dependientes a todas las secciones que las usan.
- Cada variable debería tener el mínimo alcance posible. Por ejemplo, un objeto local puede ser declarado justo antes de su primer uso.
- Haz los objetos inmutables cuando sea relevante.
- Haz al código leíble usando espacios, tanto horizontales como verticales. Por ejemplo, alineando estructuras relacionadas y usando una línea vacía para separar dos secciones.
- Haz al código semi-documentable escogiendo nombres descriptivos (pero relativamente cortos) para los objetos, tipos, funciones, etc.
- Si necesitas una sección anidada, crea una función.

- Crea tus funciones cortas y enfocadas en una sola tarea. El viejo límite de 24 líneas aún aplica. A pesar que los tamaños de las pantallas han cambiado, nada ha cambiado en la cognición humana desde la década de los sesenta.
- Las funciones deben tener pocos parámetros (cuatro es buen límite superior). Esto no restringe los datos comunicados a las funciones: agrupando parámetros relacionados en un objeto beneficia desde sus invariantes y ahorra razonamiento, tales como su coherencia y consistencia.
- En general, cada unidad de código, desde un bloque hasta una biblioteca, debería tener una interface rala. Menos comunicación reduce el razonamiento requerido. Esto significa que los getters que regresan estados internos son una responsabilidad –no pidas a un objeto la información que ya tiene–. En otras palabras, la encapsulación es todo sobre interfaces limitadas.
- Para poder preservar las clases invariantes, el uso de setters no debería ser recomendada, debido a que los setters tienden a permitir invariantes que gobiernan el estado de un objeto hacia su ruptura.

Conforme se razone sobre la correctitud, argumentar sobre tu código te ofrece entendimiento sobre él. Comunica sus descubrimientos para el beneficio de todos.

CONSEJO 16 DE 97

Un comentario acerca de los comentarios

Por Cal Evans · Traducción: Espartaco Palma

En mi primera clase de programación en la universidad, el profesor nos entregó dos hojas de codificación BASIC. En el pizarrón, se leía la asignatura: “Escribir un programa para ingresar y promediar 10 puntuaciones de bolos”. A continuación, el profesor salió de la habitación. ¿Qué tan difícil puede ser? No recuerdo mi solución final, pero estoy seguro que tenía un bucle FOR/NEXT en él y no podía haber sido de más de 15 líneas de longitud en total. Las hojas de codificación –para los niños que leen esto, sí, solíamos escribir el código a mano antes de ingresarlo a la computadora– permitían alrededor de 70 líneas de código cada una. Estaba confundido sobre por qué el maestro nos había dado dos hojas. Debido a que mi manuscrito había sido atroz, usé la segunda en transcribir mi código muy cuidadosamente, esperando obtener un par de puntos extras por el estilo.

Para mi sorpresa, cuando me regresaron la asignatura, al inicio de la siguiente clase, obtuve una calificación apenas aprobatoria. (Sería un presagio para mí el resto del tiempo en la universidad). Garabateado en la parte superior de mi cuidadosamente copiado código: “¿Sin comentarios?”.

No era suficiente que el profesor y yo supiéramos lo que se suponía haría el programa. Parte de los puntos de la asignatura era enseñarme que mi código debía explicarse por sí mismo al programador después de mí. Es una lección que no he olvidado.

Los comentarios no son malignos. Son necesarios en la programación tanto como los constructos más básicos de ramificaciones o ciclos. Los lenguajes más modernos tienen una herramienta similar a javadocs que analiza comentarios con el formato adecuado para construir automáticamente la documentación del `API`. Esto es un buen comienzo, pero no es suficiente. Dentro de tu código debería haber explicaciones acerca de lo que se supone que está haciendo. Codificar con el viejo adagio: “Si fue difícil de escribir, debe ser difícil de leer”, hace un pobre favor a tu cliente, tu empleador, tus colegas o tu propio futuro.

Por otro lado, puedes irte demasiado lejos con tus comentarios. Asegúrate de que clarifican el código, pero no lo obscurecen. Espolvorea tu código con comentarios relevantes explicando qué debe realizar. El comentario principal debería darle a cualquier programador suficiente información para usarlo sin tener que leerlo, mientras que los comentarios en línea deberían asistir al siguiente desarrollador que lo arregle o lo extienda.

En un trabajo estuve en desacuerdo con una decisión de diseño hecha por mis superiores. Por intentar ser sarcástico, como suelen ser los programadores jóvenes, copié el texto del correo en el cual se me instruía a usar su diseño en el bloque del comentario principal del archivo. Sucedió que los administradores de esta tienda en particular revisaron el código cuando lo envié. Fue mi primera introducción al término “despido por límite de profesión”.

Comenta sólo lo que el código no dice

Por Kevlin Henney · Traducción: Espartaco Palma

La diferencia entre teoría y práctica es más grande en la práctica que en la teoría –una observación que aplica a los comentarios–. En teoría, la idea general de comentar código suena como algo útil: ofrece al lector detalles, una explicación de lo que está pasando. ¿Qué podría ser más útil que ser útil? En la práctica, sin embargo, los comentarios frecuentemente se convierten en una plaga. Así como otras formas de escritura, existen habilidades para escribir buenos comentarios. Mucho de esa habilidad es saber cuándo no escribirlos.

Cuando el código está mal formado, los compiladores, intérpretes y otras herramientas se aseguran de objetar. Si el código es, de algún modo, funcionalmente incorrecto, las revisiones, los análisis estáticos, las pruebas y el uso diario en un ambiente de producción eliminará muchos de los errores. ¿Qué me dices de los comentarios? En The Elements of Programming Style, Kernighan y Plauger notaron que "un comentario tiene valor de cero (o negativo) si es erróneo". Y, sin embargo, tales comentarios ofrecen poco y sobreviven en un **código base** de una manera que los errores de codificación nunca pueden. Proporcionan una fuente constante de distracción y desinformación, un lastre sutil pero constante en el pensamiento de un programador.

¿Qué hay con los comentarios que no están técnicamente mal, pero no agregan valor al código? Son ruido. Los comentarios que parlotean el código no ofrecen algo extra al lector –decir algo una vez en código y otra vez en lenguaje natural no lo hace más verdadero o más real–. El código comentado no es código ejecutable, por lo que no tiene un efecto útil para el lector, ni en tiempo de ejecución. También se vuelve rancio fácilmente. Los comentarios relacionados a la versión y el código comentado tratan de abordar preguntas sobre las versiones y la historia. Estas preguntan ya han sido respondidas, de forma más eficiente, por las herramientas de control de versiones.

Una prevalencia de comentarios ruidosos e inconsistentes en el código base anima a los programadores a ignorar todos los comentarios, ya sea saltándolos o tomando medidas activas para ocultarlos. Los programadores tienen muchos recursos y le darán vuelta a cualquier cosa que se perciba como dañino: plegando los comentarios; cambiando el esquema de color, así los comentarios y el color de fondo se igualan; creando scripts para filtrar comentarios. Para salvar el código base de las malas aplicaciones de la ingenuidad del programador, y reducir el riesgo de pasar por alto cualquier comentario de valor genuino, los comentarios deberían ser tratados como si fueran código. Cada comentario debería agregar algo de valor al lector, de otro modo es un desperdicio que debería ser removido o reescrito.

¿Qué lo califica como valioso? Los comentarios deberían decir algo que ¿el código no hace y no puede decir. Un comentario que explica lo que ¿una pieza de código ya debería decir es una invitación para cambiar la

¿estructura del código o las convenciones de codificación para que hable ¿por sí mismo. En vez de compensar la pobreza en el nombre de los ¿métodos o de las clases, renómbralos. En vez de comentar secciones en ¿funciones largas, extrae las funciones pequeñas cuyos nombres capturen ¿las intenciones de las anteriores partes. Intenta expresar tanto como ¿sea posible a través del código. Cualquier déficit entre lo que puedes ¿expresar en código y lo que deseas expresar en su totalidad se ¿convierte en un candidato plausible para un comentario útil. Comenta lo ¿que el código no puede decir, no lo que el código no dice.

Aprendiendo continuamente

Por Clint Shank · Traducción: Espartaco Palma

Vivimos en tiempos interesantes. Conforme el desarrollo se distribuye en todo el mundo, se aprende que hay muchas personas capaces de hacer tu trabajo. Necesitas seguir aprendiendo para seguir siendo comercializable. De lo contrario, te convertirás en dinosaurio, atrapado en el mismo trabajo hasta que, un día, no serás necesario o tu trabajo será subcontratado con algún recurso más barato

Entonces, ¿qué hacer al respecto? Algunos empleadores son lo suficientemente generosos para proveer formación para ampliar tus habilidades. Otros pueden no ser capaces de ahorrar el tiempo o el dinero para entrenarte. Para jugar a la segura, necesitas tomar responsabilidad de tu propia educación.

Aquí hay una lista de las ideas para mantenerte en aprendizaje. Muchas de se pueden encontrar en Internet de forma gratuita:

- Lee libros, revistas, blogs, feeds de twitter y sitios web. Si quieres profundizar en un tema, considera unirte a una lista de correo o grupos de noticias
- Si realmente quieres estar inmerso en una tecnología, pon las manos en ello y escribe algún código.
- Trata siempre de trabajar con un mentor, sentirse el mejor puede dificultar tu educación. Aunque puedes aprender algo de cualquiera, puedes aprender mucho más de alguien más inteligente o más experimentado que tú. Si no puedes encontrar un mentor, considera seguir adelante.
- Utiliza mentores virtuales. Encuentra autores y desarrolladores en la web que realmente te gusten y lee todo lo que han escrito. Inscríbete en sus blogs.
- Conoce sobre los frameworks y bibliotecas que usan. Saber cómo funciona algo te hace saber cómo usarlo mejor. Si son de software libre, estás de suerte. Usa el depurador para ir paso a paso por el código para ver qué hay tras el telón. Podrás ver el código escrito y revisado por personas realmente inteligentes.
- Cada vez que cometas un error, arregles un error o estés en un problema trata de entender qué pasó. Es probable que alguien más haya tenido el mismo problema y haya escrito sobre él en algún lugar de la web. Google es útil en este caso.
- Una buena manera de aprender algo es enseñando o hablando sobre eso. Como la gente está para escucharte y te hará preguntas, estarás motivado a aprender. Intenta un “almuerza y aprende” en el trabajo, un grupo de usuarios o con conferencias locales.
- Inicia o únete a un grupo de estudio (a la comunidad de patrones) o a un grupo local de usuarios del lenguaje, tecnología o disciplina en la que estés interesado.
- Asiste a conferencias. Y si no puedes ir, muchas conferencias ponen sus charlas en línea gratuitamente.

> ¿Tienes un largo trayecto de la casa al trabajo? Escucha podcasts.
>
> ¿Alguna vez has ejecutado las herramientas de análisis estático sobre tu **código base** o has mirado en las advertencias de tu `IDE`? Comprende qué están reportando y por qué.
>
> Sigue la recomendación de The Pragmatic Programmer y aprende un nuevo lenguaje cada año. Al menos aprenderás una nueva tecnología o herramienta. El diversificar te dará ideas que puedes usar en tu pila tecnológica actual.
>
> No todo lo que aprendas tiene que ser sobre tecnología. Aprende el dominio de lo que estás trabajando, así puedes comprender mejor los requerimientos y ayudar a resolver el problema del negocio. Aprender a ser más productivo – cómo trabajar mejor – es otra buena opción.
>
> Vuelve a la escuela.

Sería bueno tener la capacidad que Neo tenía en The Matrix y simplemente descargar en tu cerebro la información que necesitas. Pero no podemos, por lo que requerirá un compromiso de tiempo. No tienes que gastar cada hora de vigilia aprendiendo. Un poco de tiempo, por ejemplo semanalmente, es mejor que nada. Existe (o debería haber) una vida fuera del trabajo.

La tecnología cambia rápidamente. No te quedes atrás.

La conveniencia no es una -bilidad

Por Gregor Hohpe · Traducción: Espartaco Palma

Mucho se ha dicho acerca de la importancia y desafíos al diseñar una buena `API`. Es difícil *hacerlo bien* la primera vez y es incluso más difícil cambiarlo después. Algo así como la crianza de niños. La mayoría de los programadores experimentados han aprendido que una buena API sigue un nivel consistente de abstracción, exhibe consistencia y simetría, y forma el vocabulario para un lenguaje expresivo. Por lo tanto, estar consciente de los principios guía no se traduce automáticamente en un comportamiento adecuado. Comer dulces es malo para ti.

En vez de predicar desde las alturas, quiero tomar una "estrategia" específica de diseño de API, una que me encuentro una y otra vez: el argumento de conveniencia. Comienza típicamente con los siguientes "puntos de vista":

> No quiero que otras clases tengan que hacer dos llamadas separadas para hacer una cosa.
>
> ¿Por qué debería hacer otro método si es casi igual que éste? Sólo agregaré un switch sencillo.
>
> Mira, es muy fácil: si el segundo parámetro de cadena termina con ".txt", el método automáticamente asume que el primer parámetro es el nombre de archivo, por lo que no necesito realmente dos métodos.

Aunque sea bien intencionado, tales argumentos son propensos a disminuir la **legibilidad** del código al usar el API. Una invocación de método como esta:

```
parser.processNodes(text, false);
```

no tiene virtualmente algún significado si no sabemos la implementación, o al menos consultar la documentación. Este método fue probablemente diseñado para la comodidad del implementador como un opuesto de la conveniencia de quien llama. "No quiero que quien hace la llamada tenga que hacer dos llamadas separadas" se traduce en: "no quería codificar dos métodos separados". No hay nada fundamentalmente malo con la conveniencia si tiene intención de ser el antídoto del tedio, falta de idea o incomodidad. Sin embargo, si pensamos más cuidadosamente en ello, el antídoto para esos síntomas es la eficiencia, consistencia y elegancia, no necesariamente la conveniencia. Se supone que el API oculta la complejidad subyacente, podemos esperar de manera realista que un buen diseño de API requiere algo de

esfuerzo. Un solo método largo podría ser ciertamente más conveniente de escribir que un bien pensado conjunto de operaciones, pero ¿sería fácil de usar?

La metáfora del API como un lenguaje puede guiarnos hacia mejores decisiones de diseño en estas situaciones. Un API debe proporcionar un lenguaje expresivo, lo cual nos da en el siguiente nivel suficiente de vocabulario para preguntar y responder preguntas útiles. Esto no implica que debería proveer exactamente un método o verbo por cada pregunta que valga la pena. Un vocabulario diverso nos permite expresar matices de significado. Por ejemplo, preferimos decir `correr` en vez de `caminar(true)`, a pesar de que podría ser visto como esencialmente la misma operación, sólo ejecutada en una velocidad distinta. Un vocabulario API consistente y bien pensado hace expresivo y fácil de entender el código del siguiente nivel. Más importante aún, un vocabulario que pueda ser mejorado permite a otros programadores usar el API de formas que quizás no habías anticipado –¡de hecho, una gran conveniencia para los usuarios del API!–. La próxima vez que estés tentado a agrupar unas cuantas cosas en un método API, recuerda que el idioma inglés no tiene una palabra para MakeUpYourRoomBeQuietAndDoYourHomeWork (LimpiaTuCuartoSeCalladoyHazTuTarea), a pesar de que parece muy conveniente para una operación tan frecuentemente solicitada.

Implementa rápido y con frecuencia

Por Steve Berczuk · Traducción: Espartaco Palma

Depurar el proceso de implementación e instalación suele posponerse hasta que se acerca el final del proyecto. En algunos proyectos, la escritura de herramientas de instalación es delegada a un ingeniero de entrega, quien asume la tarea como un "mal necesario". Las revisiones y demostraciones son realizadas a partir de un ambiente hecho a mano para asegurarse de que todo funciona. El resultado es que el equipo no obtiene la experiencia en el proceso de implementación o sobre el ambiente de implementación hasta que quizás es demasiado tarde para hacer los cambios.

El proceso de instalación/implementación es lo primero que ve el cliente, y un proceso simple de instalación/implementación es el primer paso para tener un ambiente de producción fiable (o, al menos, fácil de depurar). El software implementado es lo que el cliente usará. El no garantizar que la implementación configura la aplicación correctamente hará que el cliente tenga preguntas antes de que use tu software exhaustivamente.

Iniciar tu proyecto con un proceso de instalación te dará tiempo para evolucionar el proceso conforme se vaya moviendo en el ciclo de desarrollo del producto y la posibilidad para realizar cambios al código de la aplicación para que la instalación sea más fácil. Ejecutar y probar el proceso de instalación en un ambiente limpio periódicamente también provee un chequeo en el que no tendrás suposiciones en el código que se base en los ambientes de desarrollo o de prueba.

Poner la implementación al último significa que el proceso de implementación puede necesitar ser más complicado para evitar las suposiciones en el código. Lo que parece una buena idea en un `IDE`, en el cual tienes el control total de un entorno, puede hacer que un proceso de implementación sea mucho más complicado. Es mejor saber todas las ventajas y desventajas más temprano que tarde.

A pesar de que "ser capaz de implementar" desde el principio, no parece tener mucho más valor de negocio comparado con ver una aplicación ejecutándose en la computadora portátil del desarrollador, la verdad es que mientras no puedas demostrar tu aplicación en entorno final habrá un montón de trabajo que hacer antes de que puedas ofrecer un valor empresarial. Si el fundamento de poner en marcha el proceso de implementación es que es algo trivial, entonces hazlo de todos modos, ya que es a bajo costo. Si es demasiado complicado, o si hay demasiadas incertidumbres, haz lo que harías con el código de una aplicación: experimenta, evalúa y refactoriza el proceso de implementación conforme avances.

El proceso de instalación/implementación es esencial para la productividad de los clientes o de su equipo de servicio profesionales, por lo que deberías hacer pruebas y **refactorizar** este proceso sobre la marcha. Probamos y refactorizamos el código fuente de todo el proyecto. La implementación no se merece menos.

Distingue excepciones de Negocio de las excepciones Técnicas

Por Dan Bergh Johnsson · Traducción: Espartaco Palma

Hay básicamente dos razones por las que las cosas van mal en tiempo de ejecución: problemas técnicos que impiden el uso de la aplicación y la lógica del negocio que evita hacer mal uso de la aplicación. La mayoría de los lenguajes modernos, como LISP, Java, Smalltalk y C#, usan excepciones para señalar ambas situaciones. Sin embargo, las dos situaciones son tan diferentes que deberían ser tomadas por separado. Es una fuente potencial de confusión representar ambas usando la misma jerarquía de excepciones, sin mencionar la misma clase de excepciones.

Un problema técnico irresoluble puede ocurrir cuando hay un error de programación. Por ejemplo, si tratas de acceder al elemento 83 de una matriz de tamaño 17, entonces el programa está claramente fuera de control, y debería resultar en alguna excepción. La versión más sutil es llamar a alguna biblioteca de código con argumentos inapropiados, causando la misma situación dentro de la biblioteca.

Sería un error intentar resolver tú mismo estas situaciones que causaste. En vez de dejar que la excepción se eleve al nivel arquitectónico más alto y dejar que algún mecanismo general de manejo de excepciones haga lo que pueda para asegurar que el sistema está en un estado seguro, tales como deshacer una transacción, registrar en la bitácora y alertar a la gerencia, e informar (educadamente) al usuario.

Una variante de esta situación es cuando te encuentras en la "situación de biblioteca" y quien hace el llamado rompió el contrato de tu método, por ejemplo, pasando un argumento extraño o sin tener un objeto dependiente configurado correctamente. Esto va a la par con el acceso al 83vo elemento de 17: quien hace la llamada debería haber comprobado; no hacerlo es un error del programador en el lado del cliente. La respuesta correcta es lanzar una excepción técnica.

Una diferente, pero aún situación técnica, es cuando el programa no puede continuar debido a un problema en el ambiente de producción, como una base de datos que no responde. En esta situación debes asumir que la infraestructura hizo lo que pudo para resolver la situación –reparar conexiones, reintentar un número razonable de veces– y falló. Incluso si la causa es diferente, la situación para el código es similar: hay poco que puedas al respecto. Así que señalamos la situación a través de una excepción que subiremos hacia un mecanismo general de manejo de excepciones.

En contraste a esas situaciones, tenemos la situación en la cual no puedes completar la llamada por una razón de dominio lógico. En este caso nos hemos encontrado una situación que es una excepción, es decir, una inusual e indeseable, pero no un error extraño o programático. Por ejemplo, tratar de retirar dinero de una

cuenta con fondos insuficientes. En otras palabras, este tipo de situaciones es una parte del contrato, y lanzar una excepción es sólo una vía de retorno alternativa que es parte del modelo y que el cliente debería tener en cuenta y estar preparado para manejarlo. Para estas situaciones es apropiado crear una excepción específica o una jerarquía de excepción por separado, así el cliente puede manejar la situación en sus propios términos.

Mezclar excepciones técnicas y excepciones de negocios en la misma jerarquía desdibuja la distinción y confunde a quien hace la llamada sobre qué método del contrato es, qué condiciones se requiere asegurar antes de ejecutarlas y qué situaciones se supone debe manejar. Separar los casos ofrece claridad e incrementa la oportunidad de que las excepciones técnicas sean manejadas por algún framework de aplicaciones, mientras que las excepciones de dominio del negocio son consideradas y manejadas por el código del cliente.

Haz mucha práctica deliberada

Por Jon Jagger · Traducción: Espartaco Palma

La práctica deliberada no es simplemente realizar una tarea. Si te preguntas "¿porqué estoy realizando esta tarea?" y tu respuesta es "para completar la tarea", entonces no estás haciendo práctica deliberada.

Haces práctica deliberada para mejorar tu habilidad de realizar una tarea. Se trata de habilidad y técnica. La práctica deliberada significa repetición. Significa realizar la tarea con el ánimo de incrementar tu dominio de uno o más aspectos de la tarea. Significa repetir la repetición. Lentamente, una y otra vez. Hasta lograr el nivel deseado de dominio. Haces práctica deliberada para dominar la tarea, no para terminar la tarea.

El propósito principal de pagar a los desarrolladores es terminar un producto, mientras que el propósito de la práctica deliberada es mejorar tu rendimiento. No es lo mismo. Pregúntate: ¿cuánto tiempo inviertes desarrollando el producto de alguien más? ¿Cuánto desarrollándote?

¿Cuánta práctica deliberada toma el adquirir experiencia?

> Peter Norving escribe "Puede que sean 10,000 horas [...] es el número mágico".
>
> En "Leading Lean Software Development", Mary Poppendieck señala que: "A los practicantes de elite les toma un mínimo de 10 mil horas de práctica enfocada para convertirse en expertos".

La experiencia llega gradualmente con el tiempo, ¡no toda en la hora 10 mil! Sin embargo, 10 mil horas es mucho: cerca de 20 horas a la semana durante 10 años. Dado este nivel de compromiso podrías estar preocupado de no ser material experto. Lo eres. La grandeza es, en gran medida, una cuestión de elección consciente. Tu elección. Las investigaciones realizadas durante las dos últimas décadas han demostrado que el factor principal de adquisición de experiencia es el tiempo dedicado a realizar práctica deliberada. La habilidad innata no es el factor principal.

> Mary: "Hay un consenso general entre investigadores de rendimiento experto de que el talento natural no cuenta más que el esfuerzo; puedes tener una mínima cantidad de habilidad natural para iniciar en un deporte o profesión. Después de eso, la gente que es excelente es la que trabaja más duro".

No tiene mucho sentido la práctica deliberada en algo ya eres un experto. La práctica deliberada significa practicar algo en lo que no eres bueno.

> Peter: "La clave [para desarrollar experiencia] es la práctica deliberada: no sólo haciéndolo una y otra vez, pero sí retándote a ti mismo con una tarea que está más allá de tu capacidad actual, intentándolo, analizando tu rendimiento mientras y después de hacerlo, y corrigiendo cualquier error".
>
> Mary: "La práctica deliberada no significa hacer algo en lo que ya eres bueno; significa retarte a ti mismo, haciendo algo en lo que no eres bueno. Esto no es necesariamente divertido".

La práctica deliberada es acerca del aprendizaje. Acerca del aprendizaje que te cambia; del aprendizaje que cambia tu comportamiento. Buena suerte.

Lenguajes Específicos del Dominio (DSL)

Por Michael Hunger · Traducción: Espartaco Palma

Cada vez que escuches una discusión de expertos de cualquier dominio, ya sean jugadores de ajedrez, maestros de jardín de niños o agentes de seguros, notarás que su vocabulario es un poco diferente del lenguaje diario. Es parte de los Lenguajes Específicos del Dominio (`DSL`). Un dominio específico tiene un vocabulario especializado para describir cosas que son particulares de ese dominio.

En el mundo del software, los DSL tratan sobre expresiones ejecutables en un lenguaje específico de un dominio con un limitado vocabulario y gramática que es legible, entendible y –afortunadamente– escribible por expertos del dominio. Los DSL dirigidos a desarrolladores de software o científicos han estado por aquí desde hace un largo tiempo. Por ejemplo, el "pequeño lenguaje" de Unix encontrado en archivos de configuración y los lenguajes creados con el poder de macros de LISP son de los más viejos ejemplos.

Los DSL son comúnmente clasificados como internos o externos:

> Los DSL internos son escritos en un lenguaje de programación de propósito general, cuya sintaxis se ha inclinado a parecerse más al lenguaje natural. Es más fácil para los lenguajes que ofrecen azúcar sintáctica y posibilidades de formato (ej. Ruby y Scala) que para otros que no lo hacen (ej. Java). Muchos DSL internos envuelven `API` existentes, bibliotecas o código de negocio para proveer un contenedor con un acceso más alucinante a sus funcionalidades. Son ejecutados con sólo correrlos. Dependiendo en la implementación y el dominio, son usados para construir estructuras de datos, definir dependencias, ejecutar procesos o tareas, comunicarse con otros sistemas o validar entradas de usuario. La sintaxis de un DSL interno están contenidas en el lenguaje anfitrión. Hay muchos patrones –por ejemplo, constructores de expresiones, encadenadores de métodos y anotaciones– que pueden ayudarte a doblar el lenguaje anfitrión de tu DSL. Si el lenguaje anfitrión no requiere recompilación, entonces un DSL interno puede ser desarrollado rápidamente trabajando lado a lado con los expertos del dominio.
>
> Los DSL externos son expresiones gráficas o textuales de un lenguaje, aunque los DSL textuales tienden a ser más comunes que los gráficos. Las expresiones textuales pueden ser procesadas por una cadena de herramientas que incluyen léxico, un analizador, un transformador de modelo, generadores, y cualquier otro tipo de posprocesamiento. Los DSL externos son frecuentemente leídos en modelos internos, los cuales forman los fundamentos para su posterior procesamiento. Es útil definir una gramática (por ejemplo, en EBNF). Una gramática provee un punto de partida para la generación de partes de la cadena de herramientas (por ejemplo, editor, visualizador, generador de analizadores). Para los DSL sencillos, un analizador hecho a mano podría ser suficiente; usando, por ejemplo, expresiones regulares. Los analizadores personalizados pueden llegar a ser difíciles de manejar si se espera mucho de ellos, así que tiene sentido mirar las herramientas diseñadas específicamente para trabajar con gramáticas del lenguaje; por ejemplo, openArchitectureWare, ANTlr, SableCC y AndroMDA. Es también común el definir

DSL externos como los dialectos `XML`, aunque la **legibilidad** es frecuentemente un problema; sobre todo para los lectores no técnicos.

Siempre debes tomar en cuenta la audiencia objetivo de tu DSL. ¿Son desarrolladores, administradores, clientes de negocio o usuarios finales? Tienes que adaptar el nivel técnico del lenguaje, las herramientas disponibles, ayuda de sintaxis (por ejemplo, intellisense), validación temprana, visualización y representación a tu audiencia prevista. Al ocultar detalles técnicos, los DSL pueden empoderar a los usuarios, dándoles la habilidad para adaptar los sistemas a sus necesidades sin requerir la ayuda de los desarrolladores. También puede acelerar el desarrollo debido al potencial de distribución de trabajo después de que el framework inicial está en su sitio. El lenguaje puede evolucionar gradualmente. Hay también disponibles diferentes rutas de migración para expresiones existentes y gramática.

No tengas miedo de romper cosas

Por Mike Lewis · Traducción: Espartaco Palma

Todos los que tiene experiencia en el sector indudablemente han trabajado en un proyecto en el que el **código base** era, en el mejor de los casos, precario. El sistema es factorizado pobremente y cambiar alguna cosa siempre lleva a descomponer otra característica no relacionada. Cada vez que se añade un módulo, la meta del programador es cambiar lo menos que sea posible, y contener la respiración durante cada lanzamiento. Esto es el equivalente de jugar ***Jenga*** con vigas de acero en un rascacielos, y se dirige a un desastre.

La razón por la que realizar cambios es tan destroza-nervios se debe a que el sistema está enfermo. Necesita un médico, de lo contrario su condición sólo empeorará. Ya sabes lo que está mal en tu sistema, pero tienes miedo de romper los huevos para hacer tu omelet. Un cirujano experto sabe que deben hacerse cortes para operar, pero también sabe que esos cortes son temporales y se curan. El resultado final de la operación bien vale el dolor inicial y el paciente debe sanar y estar en un mejor estado del que tenía antes de la operación.

No tengas miedo de tu código. ¿A quién le importa si algo se rompe temporalmente mientras mueves las cosas? Un miedo paralizante a los cambios es lo que tiene a tu proyecto en este estado, de entrada. Invertir el tiempo para **refactorizar** se pagará por sí mismo varias veces durante el tiempo de vida de tu proyecto. Un beneficio adicional es que la experiencia de tu equipo al lidiar con el sistema enfermo los hace expertos en saber cómo debería funcionar. Aplica este conocimiento en vez de resentirte. Trabajar con un sistema que odias es algo en lo que nadie debería gastar su tiempo.

Redefine las interfases internas, reestructura módulos, refactoriza código copiado-pegado y simplifica tu diseño reduciendo dependencias. Puedes reducir significativamente la complejidad del código eliminando “casos límite”, que, a menudo, resultan de características incorrectamente acopladas. Realiza lentamente la transición de la vieja estructura a la nueva, haciendo pruebas en el camino. Tratar de realizar una larga **refactorización** en “un gran golpe” causará suficientes problemas como para hacerte considerar abandonar todo el esfuerzo a la mitad del camino.

Sé el cirujano que no tiene miedo a cortar las partes enfermas para hacer espacio a la cura. La actitud es contagiosa e inspirará a otros a empezar en los proyecto de limpieza que han estado posponiendo. Mantén una lista de “higiene” de las tareas que el equipo siente que valen la pena para el bien general del proyecto. Convence a la administración de que, a pesar de que estas tareas podrían no producir resultados visibles, reducirán los gastos y agilizarán las futuras versiones. Nunca dejes de preocuparte por la “salud” general del código.

No seas lindo con tus datos de prueba

Por Rod Begbie · Traducción: Espartaco Palma

Se estaba haciendo tarde. Estaba tirando cosas en un repositorio de datos para probar el diseño de página en el que estaba trabajando.

Me apropié de los miembros de The Clash para los nombres de usuario. ¿Nombres de empresas? Los títulos de las canciones de Sex Pistols servirían. Ahora necesito algunos símbolos de la bolsa de valores, sólo cuatro letras en mayúsculas.

Usé las palabras de **cuatro** letras.

Parecía inofensivo. Sólo algo para divertirme, y quizás también a los otros desarrolladores el día siguiente antes de enlazarlo a fuentes de datos reales. La mañana siguiente un gerente de proyecto tomó algunas capturas de pantallas para una presentación.

La historia de la programación está llena de este tipo de cuentos de guerra. Cosas que los desarrolladores y diseñadores hicieron "y que nadie más vería", las cuales inesperadamente se vuelven visibles.

El tipo de fuga puede variar, pero, cuando sucede, puede ser mortal para la persona, equipo o compañía responsables. Los ejemplos incluyen:

> Durante una junta para revisión de estatus, un cliente hace clic en un botón que todavía no ha sido implementado. El mensaje dice: "No hagas clic en eso de nuevo, idiota". .
>
> A un programador de mantenimiento de un sistema heredado se le había dicho que añadiera un mensaje de error y decidió usar la salida de registros "detrás de la escena" existentes para lograrlo. Los usuarios repentinamente se enfrentaban con mensajes como: "¡Santos errores de base de datos, Batman!", cuando algo se descomponía.
>
> Alguien confundió las pruebas con la interfaz de administración en vivo y hace una entrada de datos "graciosos". Los clientes detectaron un "Masajeador personal con forma de Bill Gates" de $1 millón de dólares a la venta en su tienda en línea.

Como para apropiarnos del viejo adagio de "una mentira puede viajar por la mitad del mundo mientras la verdad se está poniendo los zapatos", en estas fechas y épocas una metedura de pata puede ser tuiteada y facebookeada antes de que cualquiera de los desarrolladores de la zona horaria esté despierto para hacer algo al respecto.

Incluso tu código fuente no está necesariamente libre del escrutinio. En 2004, cuando un comprimido del código fuente de Windows 2000 se abrió camino en las redes de intercambio de archivos, algunos muchachos lo revisaron en busca de profanidad, insultos y otros comentarios graciosos (el comentario // TERRIBLE HORRIBLE NO DIOS QUE MAL HACK, debo admitir, ¡se vuelve adecuado para mí de vez en cuando desde entonces!).

En resumen, cuando escribas cualquier texto en tu código –ya sea comentarios, registros, mensajes o datos de prueba– siempre pregúntate a ti mismo cómo se verá si se convierte en algo público. Esto te ahorrará, todo el tiempo, algunas caras rojas.

CONSEJO 26 DE 97

¡No ignores ese error!

Por Pete Goodliffe · Traducción: Espartaco Palma

Una tarde, estaba caminando por la calle para verme con unos amigos en un bar. No habíamos compartido una cerveza en algún tiempo y quería verlos de nuevo. Con las prisas, no miré por dónde iba. Tropecé con el borde de una esquina y caí de bruces . Bueno, me lo merecía por no poner atención, supongo.

Me dolía la pierna, pero tenía prisa por ver a mis amigos. Así que me levanté y seguí. Conforme caminaba el dolor se ponía cada vez peor. A pesar de que al inicio lo desestimaba como una conmoción, me di cuenta rápidamente de que había algo mal.

Pero me apresuré hacia el bar de todos modos. Estaba en agonía en el momento en que llegué. No tuve una gran noche, porque estaba terriblemente distraído. En la mañana fui al médico y me enteré de que me había fracturado el hueso de la espinilla. De haberme detenido cuando sentí el dolor, habría prevenido un montón del daño adicional que me causé por seguir caminando. Probablemente fue el peor día-después de mi vida.

Muchos programadores escriben código como mi desastrosa salida en la noche.

¿Error, cuál error? No va a ser grave. Honestamente. Puedo ignorarlo. Esta no es una estrategia ganadora para un código sólido. De hecho, es pura flojera (de la mala). No importa que tan poco probable creas que es un error en tu código, siempre debes revisarlo y tomarlo en cuenta. Todas las veces. No estás ahorrando tiempo si no lo haces: estás almacenando problemas potenciales en el futuro.

Reportamos errores en nuestro código de distintas formas, incluyendo:

> **Códigos de Retorno**. Pueden ser usados como valores resultantes de una función para significar “no funcionó”. Los códigos de error son bastante fáciles de ignorar. No verás nada en el código que resalte el problema. De hecho, se ha convertido en una práctica estándar ignorar algunos retornos de valores de las funciones estándares de C. ¿Qué tan frecuentemente revisas el valor de retorno de `printf`?
>
> **errno**. Es una curiosa aberración de C, un conjunto de variables globales para señalar errores. Es fácil ignorarlas, difíciles de usar y da lugar a todo tipo de problemas desagradables; por ejemplo, ¿qué pasa cuando tienes múltiples hilos llamando a la misma función? Algunas plataformas te aíslan del dolor aquí; otras no.
>
> **Excepciones**. Son una forma más soportada por los lenguajes estructurados para señalar y manipular errores. Y puedes ignorarlos. ¿O no? He visto muchos códigos como estos:

EJEMPLO DE CÓDIGO

```
try {
    // ...do something...
}
catch (...) {} // ignore errors
```

La salvación en este horrible constructo es que resalta el hecho de que estás haciendo algo moralmente dudoso.

Si ignoras un error, te haces de la vista gorda y haces de cuenta que nada ha pasado, corres un gran riesgo; así como mi pierna terminó en un peor estado por no haber dejado de caminar inmediatamente, a pesar de que conduce a una falla muy compleja, enfrenta los problemas lo antes posible. Mantén una cuenta breve.

No manejar errores conduce a:

- Código frágil. Código que se llena con errores excitantes y difíciles de encontrar.
- Código inseguro. Los crackers frecuentemente explotan los pobres manejos de errores para irrumpir en los sistemas de software.
- Estructura pobre. Si es un tedio enfrentar continuamente los errores que hay en tu código, probablemente tengas una pobre interfaz. Expresa tu interfaz de tal manera que los errores sean menos intrusivos y su manejo sea menos oneroso.

Al igual que debes comprobar todos los posibles errores en tu código, necesitas exponer todas las condiciones potenciales de error en tus interfaces. No ocultarlos, pretendiendo que tus servicios siempre funcionarán.

¿Por qué no comprobamos si hay errores? Hay un serie de excusas comunes. ¿Con cuál de ellas estás de acuerdo? ¿Cómo contrarrestar cada una?

- El manejo de errores estorba el flujo del código, haciéndolo difícil de leer y difícil de detectar en el flujo “normal” de ejecución.
- Es un trabajo extra y tengo la fecha de entrega inminente.
- Sé que esa llamada de función nunca retornará un error (`printf` siempre funciona, `malloc` siempre retorna nueva memoria); si falla tenemos problemas mayores.
- Es sólo un programa de juguete y no necesita ser escrito con un nivel digno de producción.

No sólo aprendas el lenguaje, entiende su cultura

Por Anders Norås · Traducción: Espartaco Palma

En preparatoria tuve que aprender un idioma extranjero. En ese momento pensé que siendo bueno en inglés podría arreglármelas, así que escogí dormir por tres años en las clases de francés. Unos años más tarde fui a Túnez de vacaciones. El árabe es la lengua oficial ahí y, al ser una antigua colonia francesa, el francés es también de uso general. El inglés se habla sólo en zonas turísticas. Debido a mi ignorancia lingüística, me encontraba confinado en la piscina leyendo ***Finnegans Wake***, de James Joyce, un tour de formas y lenguaje. Una mezcla lúdica de más de cuarenta idiomas, fue sorprendente, aunque una agotadora experiencia. Darme cuenta de cómo mezclar palabras y frases extranjeras le dio al autor nuevas formas de expresarse es algo que he mantenido conmigo en mi carrera como programador.

En su libro, ***The Pragmatic Programmer*** (***El Programador Pragmático***), Andy Hunt y Dave Thomas nos animan a aprender un nuevo lenguaje de programación cada año. He intentado vivir de acuerdo con su consejo y a lo largo de los años he tenido la experiencia de programar en muchos lenguajes. La lección más importante de mis aventuras como políglota es que se necesita algo más que sólo aprender la sintaxis para aprender por completo el lenguaje: necesitas entender su cultura. Puedes escribir Fortran en cualquier lenguaje, pero para aprender un lenguaje tienes que adoptarlo. No pongas excusas si tu código en C# es un largo método Main con muchos métodos de ayuda, en vez de ello, aprende por qué las clases tienen sentido. No te apenes si la pasas difícil entendiendo las expresiones lambda usadas en lenguajes funcionales, oblígate a usarlas

Una vez que hayas aprendido las trabas de un nuevo lenguaje, te sorprenderás al empezar a usar lenguajes que ya sabías de nuevas maneras. Aprendí cómo usar eficazmente los `delegates` en C# programando en Ruby, liberar todo el potencial de los `generics` de .NET me dio ideas de cómo podría hacer más útiles los `generics` de Java y LINQ hizo fácil enseñarme Scala.

También tendrás un mejor entendimiento de diseño de patrones al moverte entre diferentes lenguajes. Los programadores de C encuentran que C# y Java han más comercial el patrón iterador. En Ruby y otros lenguajes dinámicos es posible seguir utilizando el patrón visitor, pero tu implementación no se parecerá al ejemplo del libro de La Banda de los 4.

Algunos podrían argumentar que ***Finnegans Wake*** es imposible de leer, mientras que otros lo aplaudirán por su belleza estilística. Para hacer el libro una lectura un poco menos temible, hay traducciones disponibles en un lenguaje único. Irónicamente, la primera traducción fue en francés. Con la codificación es similar en muchos sentidos. Si escribes código Wake con un poco de Python, algo de Java y un toque de Erlang, tus proyectos serán un desastre. Si exploras un nuevo lenguaje para expandir tu mente y obtener ideas frescas sobre cómo puedes solucionar las cosas de manera diferente, entonces encontrarás que el código que escribes en tu tan confiable lenguaje se hace más hermoso por cada nuevo lenguaje que has aprendido.

No claves tu programa en la posición vertical

Por Verity Stob · Traducción: Espartaco Palma

Una vez escribí una parodia de un ***test*** de C++ y satíricamente sugería la siguiente estrategia de manejo de excepciones:

> IDEA CLAVE
>
> Al realizar un montón de constructos `try…catch` a través de tu **código base**, podemos, algunas veces, prevenir que nuestra aplicación aborte. Creemos que el estado resultante es "clavar el cuerpo en posición vertical".

Dejando a un lado la frivolidad, realmente estaba resumiendo una lección que recibí de Doña Amarga Experiencia. Era una clase base de nuestra aplicación, una biblioteca de C++ hecha en casa. El código había sido manoseado por muchos programadores en los últimos años. Contenían código para lidiar con todas las excepciones de escape de todo lo demás. Tomando el ejemplo de Yossarian de Catch-22, decidimos o, mejor dicho, sentimos (decidir implicaba, más bien, pensarlo que estar en la construcción de este monstruo) que una instancia de esta clase debería vivir para siempre o morir en el intento.

Al final, interconectamos múltiples manejadores de excepciones. Mezclamos excepciones estructuradas de Windows con las nativas (¿recuerdas `try…catch` en C++? Yo tampoco). Cuando las cosas se caían inesperadamente, tratábamos de llamarlas de nuevo, presionando los parámetros cada vez más fuerte. Mirando atrás, me gustaría pensar que al escribir un manejador interno de try…catch dentro de una cláusula catch de otra, una especie de conciencia se apoderó de mí para haber tomado accidentalmente la ruda ruta de las buenas prácticas en la aromática pero insalubre vía de la locura. De cualquier modo, probablemente es sabiduría retrospectiva.

No necesito decir que cualquier cosa que estuviera mal en las aplicaciones basadas en esta clase se desvanecía como víctimas de la Mafia en el muelle, sin dejar atrás algún rastro útil en las burbujas que indicara qué demonios había sucedido, a pesar de las rutinas de volcado que supuestamente grabarían el desastre. Eventualmente –un largo eventualmente– hicimos un balance de lo que habíamos hecho, y experimentamos vergüenza. Reemplazamos todo el lío con un mecanismo de informe mínimo y robusto. Pero esto fue como ver muchos accidentes en la carretera.

No te molestaré más con esto –seguramente nadie más podría haber sido tan estúpido como nosotros lo fuimos–, excepto una discusión en línea que tuve recientemente con un individuo, cuyo título académico declaró que debía saberlo mejor. Estábamos discutiendo código Java en una transacción remota. Si el código fallaba, él argumentaba, debería capturar y bloquear la excepción in situ. ("¿Y entonces qué haría con ello?", pregunté. "¿Cocinarlo para la cena?").

Citó la regla del diseñador de UI: NUNCA DEJES QUE EL USUARIO VEA UN REPORTE DE EXCEPCIÓN, como si esto resolviera el asunto, poniéndolo en mayúsculas y todo lo demás. Me preguntaba si era el responsable

del código de una de esas pantallas azules de los cajeros automáticos, cuyas fotos decoran los blogs más endebles, y había sido traumatizado permanentemente.

De cualquier modo, si llegas a verlo, asienta con la cabeza y sonríe, no le hagas caso, mientras te deslizas hacia la puerta.

No confíes en el "Aquí sucede la magia"

Por AlanGriffiths · Traducción: Espartaco Palma

Si nos fijamos en cualquier actividad, proceso o disciplina, desde lo lejano parece simple. Los gerentes sin experiencia en el desarrollo piensan que lo que hacen los programadores es sencillo, y los programadores sin experiencia en administración piensan lo mismo sobre lo que hacen los gerentes.

La programación es algo que algunas personas hacen, por algún tiempo. Y la parte difícil –pensar– es la menos visible y la menos apreciada por los no iniciados. Durante décadas ha habido muchos intentos de quitar la necesidad de esta habilidad cognoscitiva. Uno de los primeros y más memorables es el esfuerzo de Grace Hopper por hacer los lenguajes de programación menos crípticos; algunos predijeron que quitaría la necesidad de programadores especializados. El resultado (COBOL) ha contribuido a los ingresos de muchos programadores especializados durante las décadas siguientes.

La visión persistente de que el desarrollo de software se puede simplificar al quitar la programación es, para el programador que entiende de lo que se trata, obviamente ingenua. Sin embargo, el proceso mental que conduce a este error es parte de la naturaleza humana y los programadores son tan propensos a realizarlo como cualquiera.

En cualquier proyecto hay muchas cosas en las que un programador no está involucrado activamente: obtener requerimientos de los usuarios, conseguir la aprobación del presupuesto, configurar el servidor de producción, implementar la aplicación a los ambientes de QA y producción, migrar el negocio desde los viejos procesos o programas, etc.

Cuando no estás involucrado activamente en estas cosas existe una tendencia inconsciente a asumir que son sencillas y que las cosas suceden "por arte de magia". Mientras la magia siga ocurriendo todo está bien. Pero cuando –esto sucede "cuando" y no "si"– la magia se detiene, el proyecto está en problemas.

He conocido proyectos que pierden semanas de desarrollo porque nadie entiende cómo se confía en la versión "correcta" de un DLL que está siendo cargado. Cuando las cosas empezaron a fallar intermitentemente los miembros del equipo miraban a cualquier otra parte antes de que alguien notara que una versión "equivocada" del DLL había sido cargada.

Otro departamento estaba funcionando sin problemas, proyectos enviados a tiempo, no más sesiones de depuración nocturna, no arreglos de emergencia. Tan tranquilamente, de hecho, que la Alta Gerencia decidió que las cosas "corrían por sí mismas" y lo podría hacer sin el administrador de proyectos. En los siguientes seis meses los proyectos en el departamento se veían tan bien como en el resto de la organización: retrasados, con errores y siendo parchados continuamente.

No tienes que entender toda la magia que sucede en tu proyecto, pero no está de más entender algo de ella, o apreciar a aquellos que entienden las partes que tú no.

Más importante, asegúrate de que cuando la magia se detenga, pueda ser iniciada de nuevo.

No te repitas

Por Steve Smith · Traducción: Espartaco Palma

De todos los principios de programación, *No te Repitas* (***Don't Repeat Yourself***, DRY) es quizás uno de los fundamentales. El principio fue formulado por Andy Hunt y Dave Thomas en ***The Pragmatic Programmer*** y subyace a muchas otras bien conocidas buenas prácticas y diseños de patrones en software. El desarrollador que aprende a reconocer la duplicación y entiende cómo eliminarla, a través de una abstracción práctica y apropiada, puede producir código mucho más limpio que quien infecta continuamente la aplicación con repetición innecesaria.

La duplicidad es un desperdicio

Cada línea de código que va en una aplicación se debe mantener y es una fuente potencial de futuros errores. La duplicación infla innecesariamente el **código base**, dando lugar a más oportunidades para los errores y agregando complejidad accidental al sistema. El atasco que la duplicación agrega al sistema también hace más difícil para los desarrolladores que trabajan con el sistema el completo entendimiento del sistema entero, o de tener la certeza de que los cambios realizados en un lugar no necesitan también ser hechos en otros lugares que duplican la lógica de lo que se está trabajando. DRY requiere que "cada pieza de conocimiento debe tener una representación única, inequívoca y autorizada en el sistema".

Cambiar repetición en llamadas de procedimiento por automatización

Muchos de los procesos en el desarrollo del software son repetitivos y fácilmente automatizados. El **principio DRY** se aplica en estos contextos tan bien como en el código fuente de la aplicación. Las pruebas manuales son lentas, propensas al error y difíciles de repetir, por lo que, si es posible, deberían usarse los conjuntos de **pruebas automatizadas**. Integrar software puede tomar mucho tiempo y ser propenso al error si se hace manualmente, por lo que el proceso de construcción deberá ser ejecutado tan frecuente como sea posible, idealmente en cada ***check-in***. Donde sea que existan esos dolorosos procesos que puedan ser automatizados, deben ser automatizados y estandarizados. La meta es asegurarse de que sólo hay una manera de llevar a cabo la tarea, y que ésta sea lo menos dolorosa posible.

Cambiar repetición en la lógica por abstracción

La repetición en la lógica puede tomar muchas formas. Copiar-pegar lógica de un `if-then` o `switch-case` es uno de los casos más comunes de detectar y corregir. Muchos **patrones de diseño** tiene la meta específica de reducir o eliminar la duplicación en la lógica de una aplicación. Si un objeto usualmente requiere que varias cosas sucedan antes de que pueda ser utilizado, esto se puede lograr con una `Abstract Factory` o `Method Factory`. Si un objeto tiene muchas variaciones posibles en su comportamiento, estos comportamientos pueden ser inyectados con el patrón de Estrategia en vez de largas estructuras `if- then`. De hecho, la formulación de patrones de diseño es un intento de reducir la duplicación del esfuerzo necesario

para resolver problemas comunes y discutir dichas soluciones. Adicionalmente, DRY puede ser aplicado a estructuras, tales como esquemas de base de datos, resultando en la normalización.

Una cuestión de principio

Otros principios de software también están relacionados con DRY. El principio "Uno y Sólo Uno", el cuál aplica al comportamiento funcional del código, puede ser pensado como un subconjunto de DRY. El principio "Abierto/Cerrado", el cual estipula que "las entidades de software deben estar abiertas para la extensión, pero cerradas para la modificación", sólo funciona en la práctica cuando se sigue el DRY. Del mismo modo, el bien conocido Principio de la **Responsabilidad Única** (SPR), que requiere que una clase tenga "una única razón para cambiar", se basa en DRY.

Cuando se sigue en estructura, lógica, procesos y funciones, el principio provee una guía fundamental para los desarrolladores de software y ayuda a la creación de aplicaciones más simples, más fáciles de mantener y de alta calidad. Si bien hay escenarios en los cuales la repetición puede ser necesaria para cumplir con el índice de rendimiento u otros requerimientos (por ejemplo, desnormalización en base de datos), esto debería ser usado sólo donde aplique directamente un problema real en vez de uno imaginario.

¡No toques ese código!

Por Cal Evans · Traducción: Espartaco Palma

Nos ha pasado a todos en algún momento. Tu código fue llevado al servidor de ***staging*** para las pruebas del sistema y el director de pruebas te lo regresa diciendo que tiene un problema. Tu primera reacción es “rápido, déjame arreglarlo, sé qué está mal”.

En un sentido más amplio, sin embargo, lo que está mal es que como desarrollador creas que deberías tener acceso al servidor de ***staging***.

En la mayoría de los ambientes de desarrollo basado en web la arquitectura puede fragmentarse así:

- Desarrollo local y pruebas unitarias en la máquina del desarrollador.
- Servidor de desarrollo, en el que se realizan las pruebas de integración, manuales o automáticas.
- Servidores de ***staging***, en el cual el equipo de Control de Calidad y los usuarios realizan las pruebas de aceptación.
- Servidor de producción.

Sí, hay otros servidores y servicios salpicados por ahí, como el control de código fuente (SCC, ***Source Code Control***) y el sistema de tickets, pero tienes la idea. Usando este modelo, un desarrollador –incluso un desarrollador experimentado– nunca debería tener acceso más allá del servidor de desarrollo. La mayor parte del desarrollo es hecho en la máquina del desarrollador usando su mezcla favorita de `IDE`, máquinas virtuales y una apropiada cantidad de magia negra para la buena suerte.

Una vez que el código se envía al SCC, ya sea automática o manualmente, debería ser pasado al servidor de desarrollo, en el cual puede ser probado y ajustado, si es necesario, para asegurarse de que todo funciona. A partir de este momento, sin embargo, el desarrollador es un espectador en el proceso.

El director de ***staging*** debería empaquetar y desplegar el código al servidor de ***staging*** del equipo de Control de Calidad. Así como los desarrolladores deberían no tener acceso a nada más allá del servidor de desarrollo, el equipo de Control de Calidad y los usuarios no tienen necesidad de tocar nada en el servidor de desarrollo. Si está listo para las pruebas de aceptación, libéralo y envíalo, no pidas al usuario “mirar algo muy rápido” en el servidor de desarrollo. Recuerda, a menos que estés codificando el proyecto tú solo, que otras personas tienen código ahí y podría no estar listo para lo mire el usuario. El encargado de liberaciones es la única persona que debería tener acceso a ambos.

Bajo ninguna circunstancia –nunca, en lo absoluto– debe un desarrollador tener acceso al servidor de producción. Si hay algún problema, el personal de soporte debería solucionarlo o requerir que lo arreglen. Después de enviarlo al SCC, ellos pasarán un parche desde ahí. Algunos de los mayores desastres de programación de los que he sido parte han tenido lugar porque alguien ***ejeeemmmyoeejeeemm*** violó esta última regla. Si está descompuesto, producción no es el lugar para arreglarlo..

CONSEJO 32 DE 97

Encapsula Comportamiento, no sólo Estado

Por Einar Landre · Traducción: Espartaco Palma

En la teoría de sistemas, el contenimiento es uno de los más útiles constructos cuando se está tratando con sistemas de estructuras muy grandes y complejas. En la industria de software el valor del contenimiento o encapsulación es bien entendido.

Los módulos y paquetes resuelven las necesidades a gran escala de la encapsulación, mientras que las clases, subrutinas y funciones resuelven los aspectos más granulares en la materia. A través de los años he descubierto que las clases parecen ser uno de los constructos de encapsulación más difíciles que los desarrolladores entiendan. Es común encontrar una clase con sólo un método principal con 3 mil líneas de código, o una clase con sólo método `set` y `get` para sus atributos primitivos. Estos ejemplos demuestran que el desarrollador involucrado no ha entendido por completo el pensamiento orientado a objetos, fallando en tomar ventaja del poder de los objetos como constructos de modelaje. Para los desarrolladores familiarizados con los términos POJO (***Plain Old Java Object***) y POCO (***Plain Old C# Object*** o ***Plain Old CLR Object***), éste fue el intento para regresar a lo más básico de OO como el paradigma modelo, los objetos son planos y sencillos, pero no tontos.

Un objeto encapsula ambos; estado y comportamiento, donde el comportamiento es definido por el estado actual. Considera un objeto puerta. Éste tiene 4 estados: cerrado, abierto, cerrando, abriendo. Ofrece dos operaciones: abrir y cerrar. Dependiendo del estado, las operaciones de abrir y cerrar se comportarán de forma diferente. Esta propiedad inherente de un objeto hace que el proceso de diseño conceptualmente simple. Esto se resume en dos tareas sencillas: localización y delegación de responsabilidad a los diferentes objetos, incluyendo los protocolos de la interacción entre objetos.

Cómo funciona en la práctica se ilustra mejor con un ejemplo. Digamos que tienes tres clases: ***Customer*** (Cliente), ***Order*** (Orden) e ***Item***. El objeto ***Customer*** es marcador de posición natural para el límite de crédito y las reglas de validación. Un objeto ***Order*** sabe sobre su Customer asociado, y su operación `addItem` delega la validación del crédito actual llamando al método `Customer.validaCredito(item.precio())`. Si la poscondición del método falla, una excepción puede ser enviada y la compra cancelada.

Los desarrolladores menos experimentados en orientación a objetos podrían decidirse a envolver todas las reglas de negocio en un objeto frecuentemente referido como ***orderManager*** u ***OrderService***. En este diseño, ***Order***, ***Customer*** e ***Item*** son tratados como algo más que tipos de registros. Toda la lógica es factorizada desde las clases y unidas en un método largo y procedural con un montón de constructos internos `if-the-else`. Estos métodos se rompen con facilidad y son casi imposibles de mantener. ¿La razón? La encapsulación está rota.

Así que, al final, no rompas la encapsulación y usa el poder de tu lenguaje de programación para mantenerla.

Los números de punto flotante no son reales

Por Chuck Allison · Traducción: Espartaco Palma

Los números de punto flotante no son "números reales" en el sentido matemático, a pesar de que son llamados ***reales*** en algunos lenguajes de programación, como Pascal y Fortran. Los números reales tienen una precisión infinita y son, por lo tanto, continuos y sin pérdidas; los números de punto flotante tiene precisión limitada, por lo que son finitos, y son recordados como enteros "con mal comportamiento", porque no son están espaciados uniformemente a través de su espacio de distribución.

Para ilustrarlo, asigna 2147483647 (el número más grande en un entero de 32 bits) a una variable `float` de 32 bits (digamos x) e imprímelo. Verás 2147483648. Ahora imprime `x - 64`. Aún 2147483648. Ahora calcula `x - 65` y ¡obtendrás 2147483520! ¿Por qué? Porque la separación entre flotantes adyacentes en ese rango es de 128 y las operaciones de punto flotante se redondean al punto flotante más cercano.

Los números de punto de flotante de la IEEE son números de precisión fija basados en notación científica de base 2: $1.d_1d_2...d_{p-1} \times 2^e$, donde p es la precisión (24 para `float`, 53 para `double`). El espaciamiento entre dos números consecutivos es 2^{1-p+e}, lo cual puede ser aproximado con seguridad a ε|x|, donde ε es el épsilon de la máquina (2^{1-p}).

Conocer el espaciamiento en los vecinos de un número de punto flotante puede ayudarte a evitar errores numéricos clásicos. Por ejemplo, si estás realizando un cálculo iterativo, como buscar la raíz de una ecuación, no tiene sentido buscar una precisión más grande que el sistema numérico puede darte en la cercanía de la respuesta. Asegúrate que la tolerancia que pides no es menor que el espaciado ahí; de otro modo harás un bucle infinito.

Debido a que los números de punto flotante son aproximaciones de los números reales, inevitablemente hay un pequeño error presente. Este error, llamado redondeo, puede llevarnos a errores sorpresivos. Por ejemplo, cuando sustraes números cercanamente iguales, los dígitos más significativos se cancelan entre sí, entonces lo que era el dígito menos significativo (donde reside el error de redondeo) es promovido a la posición más significativa en el resultado de punto flotante, contaminando esencialmente cualquier cómputo relacionado (un fenómeno conocido como ***smearing***). Necesitas mirar muy de cerca tus algoritmos para prevenir esa cancelación catastrófica. Para ilustrarlo, considera resolver la ecuación $x^2 - 100000x + 1 = 0$ con la fórmula cuadrática. Como los operandos en la expresión _-b + sqrt(b^2

> 4)_ son cercanamente iguales en magnitud, puedes en su lugar computar la raíz $r_1 = -b - sqrt(b^2 -4)$, y entonces obtener $r_2 = 1/r_1$, como en cualquier ecuación cuadrática, $ax^2 + bx + c = 0$, entonces la raíz satisface $r_1r_2 = c /a$.

El ***smearing*** puede ocurrir incluso en formas más sutiles. Supón una librería que ingenuamente computa e^x con la fórmula _1 + x

> $x^2/2 + x^3/3! +$_ ... Esto funciona bien para una x positiva, pero considera qué pasa cuando x es un número negativo grande. Los términos impares potenciados resultan en un número positivo grande y sustrayendo las magnitudes de pares potenciados ni se verán afectados en el resultado. El problema aquí es que el redondeo en los grandes términos positivos está a un dígito de posición de la más grande significancia que la verdadera respuesta. ¡La respuesta difiere hacia positivo infinitamente! La solución aquí también es simple: para una x negativa, computa $e^x = 1/ e^{|x|}$.

No podemos irnos sin decir que no deberías usar números de punto flotante para aplicaciones financieras, para eso son las clases decimales en lenguajes como Python y C#. Los números de punto flotante son para un cómputo científico eficiente. Pero la eficiencia es inútil sin precisión, ¡así que recuerda la fuente de los errores de redondeo y codifica en consecuencia!

Cumple tus ambiciones con Software Libre

Por Richard Monson-Haefel · Traducción: Espartaco Palma

Hay una alta probabilidad de que no estés desarrollando software en tu trabajo para cumplir tus más ambiciosos sueños. Quizás estás desarrollando software para una gran compañía de seguros cuando te gustaría estar trabajando en Google, Apple, Microsoft o tu propia ***start-up***, desarrollando la próxima “gran cosa”. Nunca vas a llegar a donde quieres desarrollando software para sistemas que no te importan.

Afortunadamente, hay una respuesta a tus problemas: software libre. Hay miles de proyectos de software libre por ahí, muchos de ellos muy activos, los cuales ofrecen cualquier tipo de experiencia de desarrollo de software que puedas desear. Si amas la idea de desarrollar un sistema operativo, ve y ayuda con alguno. Si deseas trabajar con software de música, animación, criptografía, robótica, juegos de PC, juegos masivos en línea, teléfonos móviles o lo que sea, puedes estar casi seguro de que encontrarás, al menos, un proyecto de software libre dedicado a ese interés.

Por supuesto que no hay almuerzos gratis. Tienes que estar dispuesto a dar tu tiempo libre porque probablemente no puedas trabajar en el un videojuego de software libre en tu trabajo, aún tienes responsabilidad con tu empleador. Adicionalmente, muy pocas personas hacen dinero contribuyendo con proyectos de software libre. Debes estar dispuesto a renunciar a una parte de tu tiempo libre (menos tiempo jugando videojuegos y mirando TV no te matará). Cuanto más trabajes en un proyecto de software libre, más rápido te darás cuenta de tus verdaderas ambiciones como programador. También es importante considerar tu contrato de empleado, algunos empleadores pueden restringir contribuciones, incluso en tu propio tiempo. Además, es necesario tener cuidado con las violaciones de las leyes de propiedad intelectual que tienen que ver con derechos de autor, patentes, marcas registradas y secretos comerciales.

El software libre provee enormes oportunidades para el programador motivado. En primer lugar, se llega a ver cómo alguien más implementa una solución que te interesa –puedes aprender mucho leyendo el código de otras personas–. En segundo lugar, se llega a contribuir con tu propio código e ideas al proyecto –no todas las ideas brillantes que tengas serán aceptadas, pero algunas podrían serlo, y aprenderás algo nuevo con sólo trabajar en soluciones y contribuir con el código–. En tercer lugar, conocerás a personas grandiosas con la misma pasión que tú por el mismo tipo de software –estas amistades pueden duran toda la vida–. En cuarto lugar, asumiendo que eres un contribuidor competente, estarás en disposición de agregar la experiencia del mundo real en la tecnología que actualmente te interesa.

Iniciar con el software libre es bastante fácil. Hay plena documentación en las herramientas que necesitas (por ejemplo, administración de código fuente, editores, lenguajes de programación, sistemas de construcción, etcétera). Primero, encuentra el proyecto en el que deseas trabajar y aprende acerca de las herramientas que utiliza. La documentación en proyectos por sí misma será una luz en muchos casos, pero esto quizás importe menos debido a que la mejor manera de aprender es investigar el código por ti mismo. Si

deseas estar involucrado, puedes ofrecer tu ayuda con la documentación. O puedes comenzar como voluntario para escribir las pruebas de código. A pesar de que esto podría no sonar excitante, la verdad es que aprendes mucho más rápido escribiendo pruebas de código para el software de otra persona como casi cualquier otra actividad en software. Escribe pruebas de código, realmente buenas pruebas de código; encuentra errores; sugiere correcciones; haz amigos; trabaja en el software que te gusta y cumple tus ambiciones.

La regla de oro del diseño de API

Por Michael Feathers · Traducción: Espartaco Palma

El diseño de `API` es difícil, particularmente los grandes. Si estás diseñando una API que va a tener cientos o miles de usuarios, tienes que pensar qué podrías cambiar en el futuro y si los cambios pueden romper el código de tu cliente. Más allá de esto, tienes que pensar cómo te afectan los usuarios de tu API. Si unas de tus clases API usa uno de sus métodos internamente, tienes que recordar que un usuario podría hacer una subclase de tu clase y sobrescribirla, y eso puede ser desastroso. No podrías ser capaz de cambiar ese método porque alguno de tus usuarios le ha dado un significado diferente. Tus futuras opciones de implementación interna están a merced de tus usuarios..

Los desarrolladores de API solucionan este problema de varias formas, pero la más fácil es bloquear el API. Si estás trabajando en Java quizás estés tentado a hacer `final` a la mayoría de tus clases y métodos. En C# podrías hacer `sealed` tus clases y métodos. Independientemente del lenguaje que estés usando, podrías estar tentado a presentar tu API a través de un singleton o usar métodos ***factory*** estáticos, así puedes defenderte de la gente que podría sobrescribir el comportamiento y usar tu código de formas que podrían restringir tus opciones más adelante. Todo esto parece razonable, pero ¿lo es realmente?

En la última década nos hemos dado cuenta gradualmente de que las pruebas unitarias son una parte importante de la práctica, pero esa lección no ha penetrado completamente la industria. La evidencia está a nuestro alrededor. Toma arbitrariamente una clase que no ha sido probada y que use un API de terceros e intenta escribir una prueba unitaria para él. La mayoría de las veces encontrarás problemas. Encontrarás que el código usando el API se pega a él como con pegamento. No hay manera de impersonalizar las clases del API para que puedas detectar las interacciones de tu código con ellos o proporcionar valores de retorno para la prueba.

Con el tiempo, esto va a mejorar, pero sólo si empezamos a ver las pruebas como un caso de uso real cuando diseñamos API. Desafortunadamente, es un poco más complicado que sólo probar nuestro código. Es aquí donde encaja la Regla de Oro del diseño de API: no es suficiente escribir pruebas del API que desarrollas; tienes que escribir pruebas unitarias para el código que usa tu API. Cuando lo haces aprendes de primera mano los obstáculos que tus usuarios tendrán que superar cuando intenten probar su código independientemente.

Cuando no hay una forma única de hacer fácil para los desarrolladores el probar el código que usa tu API, ni `static`, `final` o `sealed` son inherentemente malos constructos. A veces pueden ser útiles, pero es

importante tener en cuenta el tema de las pruebas y, para lograrlo, tienes que experimentarlo tú mismo. Una vez que lo haces, puedes enfocarlo como lo harías con cualquier otro reto de diseño.

El mito del Gurú

Por Ryan Brush · Traducción: Espartaco Palma

Cualquiera que haya trabajado en el software el tiempo suficiente ha escuchado preguntas como éstas: “estoy obteniendo una excepción XYZ. ¿Sabes cuál es el problema?”.

Aquellos que hacen la pregunta rara vez se molestan en incluir la pila de rastreo, registros de error o algún contexto que nos conduzca al problema. Al parecer creen que operas en un plano distinto, que las soluciones se te aparecen sin ningún análisis basado en evidencia. Piensan que eres un gurú.

Esperamos dichas preguntas de quienes no tienen familiaridad con el software: para ellos los sistemas pueden verse como algo mágico. Lo que me preocupa es estar viendo esto en la comunidad del software. Preguntas similares surgen en el diseño de programas, tales como: “estoy construyendo un gestor de inventarios. ¿Debo utilizar el bloqueo optimista?”. Irónicamente, la gente que hace la pregunta está mejor calificada para resolverla que el destinatario. Los interrogadores presumiblemente conocen el contexto, los requisitos y pueden leer acerca de las ventajas y desventajas de las diferentes estrategias. Sin embargo, esperan que les des una respuesta inteligente sin un contexto. Esperan magia.

Es tiempo de que la industria del software disipe este mito del gurú. Los “gurús” son humanos. Ellos aplican la lógica y el análisis sistemático de los problemas, como el resto de nosotros. Aprovechan los atajos mentales y la intuición. Considera al mejor programador que hayas conocido: en algún momento esa persona sabía menos acerca del software de lo que tú ahora mismo. Si alguien parece ser un gurú, es debido a sus años dedicados al aprendizaje y al perfeccionamiento de los procesos de pensamiento. Un “gurú” es simplemente una persona inteligente con curiosidad incesante.

Por supuesto, sigue habiendo una gran variabilidad en la aptitud natural. Muchos hackers son más inteligentes, informados y productivos de lo que alguna día puedo llegar a ser. Aun así, el desmitificar el mito del gurú tiene un impacto positivo. Por ejemplo, si trabajo con una persona más inteligente que yo, me aseguro de hacer el trabajo de campo, proveer el suficiente contexto para que esa persona pueda aplicar eficazmente sus habilidades. Quitar el mito del gurú también significa eliminar una barrera en la percepción de mejora. En vez de una barrera mágica, veo continuidad y puedo avanzar.

Por último, uno de los mayores obstáculos en el software es la gente inteligente que propaga el mito del gurú a propósito. Esto podría hacerse por ego, o como una estrategia para incrementar el valor percibido por un cliente o por su empleador. Irónicamente, esta actitud puede hacer que las personas inteligentes sean menos valiosas debido a que no contribuyen al crecimiento de sus compañeros. No necesitamos gurús. Necesitamos expertos que estén dispuestos a desarrollar a otros expertos en su campo. Hay espacio para todos nosotros.

El trabajo duro no paga

Por Olve Maudal · Traducción: Espartaco Palma

Como programador, trabajar duro muchas veces no da frutos. Puedes engañarte a ti mismo y a unos pocos colegas al creer que estás contribuyendo mucho al proyecto al pasar largas horas en la oficina. Pero, la verdad, es que trabajando menos puedes lograr más, a veces mucho más. Si tratas de estar centrado y ser "productivo" por más de treinta horas a la semana, entonces probablemente estás trabajando demasiado duro. Debes considerar reducir la carga de trabajo para ser más eficaz y hacer más cosas.

Esta frase puede parecer contraria a la intuición e incluso controversial, pero es una consecuencia directa del hecho de que la programación y el desarrollo de software, en conjunto, implican un proceso de aprendizaje continuo. A medida en que trabajas en un proyecto entenderás más sobre el dominio del problema y, con suerte, encontrarás la manera más eficaz de alcanzar tu meta. Para evitar el desperdicio de trabajo, debes permitirte tiempo para observar los efectos de lo que estás haciendo, reflexionar sobre las cosas que se ven y cambiar el comportamiento en consecuencia.

La programación profesional no suele ser como correr duro durante unos cuantos kilómetros, donde la meta puede ser vista al final de un camino pavimentado. Muchos proyectos de software son más como un largo maratón orientado; en la oscuridad, con sólo un mapa esquemático como guía. Si acabas de salir hacia una dirección, corriendo tan rápido como puedas, podrías impresionar a algunos, pero no es probable que tengas éxito. Necesitas mantener un ritmo sostenido y ajustar el curso cuando se aprende más sobre dónde te encuentras y hacia dónde te diriges.

Adicionalmente, siempre necesitarás aprender más sobre el desarrollo de software, en general, y técnicas de programación, en lo particular. Probablemente tengas que leer libros, ir a conferencias, comunicarte con otros profesionales, experimentar con nuevas técnicas de implementación y aprender acerca de potentes herramientas que simplificarán el trabajo. Como un programador profesional debes mantenerte actualizado en tu campo de especialización; al igual que se espera que los neurocirujanos y los pilotos se mantengan actualizados en sus propios campos de experiencia. Necesitas pasar tardes, fines de semana y días festivos educándote, por lo tanto, no puedes pasar tus tardes, fines de semana y días festivos trabajando tiempo extra en tu proyecto actual. ¿Realmente esperas que los neurocirujanos realicen cirugías 60 horas a la semana o que los pilotos vuelen 60 horas semanalmente? Por supuesto que no, la preparación y educación son parte esencial de su profesión.

Enfócate en el proyecto, contribuye tanto como puedas encontrando soluciones inteligentes, mejora tus habilidades, reflexiona sobre lo que estás haciendo y adapta tu comportamiento. Evitar avergonzarte, y a nuestra profesión, al comportarte como un hámster en una jaula corriendo en la rueda. Como programador profesional debes saber que tratar de estar concentrado y ser "productivo" 60 horas a la semana no es lo más sensato. Actúa como un profesional: prepárate, sé eficaz, observa, reflexiona y cambia.

¿Cómo usar un Gestor de Errores?

Por Matt Doar · Traducción: Espartaco Palma

Como sea que lo llames: bug, defecto o incluso “efecto del lado de diseño”, no hay manera de alejarse de ellos. Saber enviar un buen reporte de error y lo que se debe buscar son habilidades para mantener un proyecto que se lleve bien.

Un buen reporte de error necesita tres cosas:

- Cómo reproducir el error, lo más preciso posible, y la frecuencia con que esto hará que aparezca el error.
- ¿Qué debería haber ocurrido? Al menos en tu opinión.
- ¿Qué ocurrió realmente? Toda la información que has registrado.

La cantidad y calidad de la información reportada dice mucho acerca de quién reporta y del error mismo. Los errores con enojo o tensión (“¡esta función apesta!”) nos dice que los desarrolladores estaban teniendo un mal momento, pero no más. Un error con gran cantidad de contexto para que sea más fácil reproducirlo gana el respeto de todo el mundo, incluso si detiene una liberación.

Los errores son como un conversación, con toda la historia ahí en frente de todos. No culpes a otros o niegues la existencia del error. En vez de eso pide más información o considera qué pudiste haber olvidado.

Cambiar el estatus de un error, por ejemplo, de Abierto a Cerrado, es una declaración pública de lo que se piensa del error. Tomarse el tiempo de explicar por qué crees que el error debería estar cerrado ahorrará horas de tedio en justificarlo a directores y clientes frustrados. Cambiar la prioridad de un error es similar a las declaraciones públicas, y sólo porque es trivial no significa que alguien está dejando de usar el producto.

No sobrecargues los campos del error para tu propio propósito. Agregar “VITAL:” al campo de título de error puede hacer que sea fácil ordenar los resultados en algún informe, pero hará que eventualmente sea copiado por otros e inevitablemente será mal escrito o necesitará ser removido para su uso en algún otro informe. En vez de eso usa un nuevo valor o un nuevo campo, y documenta cómo el campo se supone debe ser usado, así otras personas no tienen que repetirlo.

Asegúrate que todos sepan cómo encontrar el error en el que se supone está trabajando el equipo. Esto se puede hacer mediante una consulta pública con un nombre obvio. Asegúrate que todos están usando la misma consulta, y no la actualices sin primero informar al equipo que estás cambiando algo en lo que todos están trabajando.

Recuerda, un error no es una unidad estándar de trabajo, como tampoco una línea de código es una unidad precisa de esfuerzo.

Mejora el código quitándolo

Por Pete Goodliffe · Traducción: Espartaco Palma

Menos es más. Es una máxima un poco trillada, pero algunas veces es cierto.

Una de las mejoras que he hecho en nuestro **código base** en las últimas semanas es eliminar trozos de él.

Hemos escrito el software siguiendo los principios de XP, incluyendo YAGNI (You Aren't Gonna Need It [No vas a necesitarlo]). La naturaleza humana es así, inevitablemente nos quedamos cortos en unos pocos lugares.

Observé que el producto estaba tomando demasiado tiempo para ejecutar ciertas tareas, tareas sencillas que deberían ser casi instantáneas. Esto era porque estaban sobreimplementadas; adornadas con campanas y silbatos adicionales que no eran requeridos, pero que en ese momento parecían una buena idea.

Simplifiqué el código, mejorando el rendimiento del producto y reduciendo el nivel de entropía global del código al quitar las características infractoras del código base. Afortunadamente, mis Pruebas Unitarias me dijeron que no había roto nada durante la operación.

Una experiencia sencilla y completamente satisfactoria.

Así que ¿por qué terminó ahí ese código innecesario? ¿Por qué un programador sintió la necesidad de escribir código adicional y cómo pasó la última revisión o el proceso entre pares? Es casi seguro que sucedió algo como esto:

> Era un poco de diversión extra y el programador quería escribirlo. (Sugerencia: escribir código porque agrega valor, no porque te divierte).
>
> Alguien pensó que podría ser necesario en el futuro, así que sintió que era mejor escribirlo ahora. (Sugerencia: esto no es YAGNI. Si no lo necesitas en este momento, no lo escribas ahora mismo).
>
> No parecía ser un gran "extra", así que era más fácil implementarlo en vez de regresar con el cliente para ver si era requerido. (Sugerencia: siempre toma más tiempo escribir y mantener código adicional. Y el cliente siempre está disponible. Una partecita extra de código se vuelve una bola de nieve en descenso con el paso del tiempo, convirtiéndose en una gran parte de trabajo que necesita ser mantenido).
>
> El programador inventó requisitos adicionales que no fueron documentados, ni discutidos para justificar la función adicional. El requerimiento era en realidad falso. (Sugerencia: los programadores no establecen los requerimientos del sistema; el cliente sí).

¿En qué trabajas ahora mismo? ¿Es todo necesario?

Instalame

Por Marcus Baker · Traducción: Espartaco Palma

No tengo el menor interés en tu programa.

Estoy rodeado de problemas y tengo una lista de cosas por hacer tan larga como mi brazo. La única razón por la que estoy en tu sitio web ahora mismo es porque he oído un poco probable rumor de que cada uno de mis problemas será eliminado por tu software. Perdóname si soy escéptico.

Si los estudios de seguimiento del globo ocular son correctos, ya he leído el título y estoy buscando un texto subrayado con color azul marcado como "descargar ahora". Como anotación al margen, si llegué a esta página con un navegador de Linux con una IP del Reino Unido, es probable que me gustaría una versión Linux desde un espejo en Europa, así que por favor no preguntes. Asumiendo que el diálogo de archivo se abre directamente, llevo la cosa a mi carpeta de descargas y sigo leyendo.

Todos nosotros realizamos constantemente análisis de costo-beneficio de lo que hacemos. Si tu proyecto cae debajo de mi umbral por un segundo, me desharé de él e iré a otra cosa. La gratificación instantánea es mejor.

El primer obstáculo es instalar. ¿No crees que sea mucho problema? Ahora ve a tu carpeta de descargas y mira alrededor. ¿Lleno de archivos .tar y .zip, verdad? ¿Qué porcentaje de esos han sido desempacados? ¿Cuántos has instalado? Si eres como yo, sólo un tercio está haciendo algo más que actuar como relleno en el disco duro.

Podría querer conveniencia a la puerta, pero no quiero que entres a mi casa sin invitación. Antes de escribir install querría saber exactamente dónde estás poniendo cosas. Es mi computadora y quiero mantenerla ordenada cuando pueda. También quiero ser capaz de eliminar tu programa al instante en el que me desencante de él. Si sospecho que eso es imposible no lo instalaré en primer lugar. Mi máquina es estable ahora y quiero que siga así.

Si tu programa se basa en `GUI` entonces quiero hacer algo simple y ver un resultado. Los ***Asistentes*** no ayudan, porque ellos hacen cosas que no entiendo. Hay probabilidad de que quiera leer o escribir un archivo. No quiero crear proyectos, importar directorios o decirte mi correo electrónico. Si todo está funcionando, ir al tutorial

Si tu software es una biblioteca, entonces seguiré leyendo tu página web buscando una guía rápida de inicio. Quiero el equivalente de un "hola, mundo" en cinco líneas sin mucho pensar con la salida descrita por tu sitio web. No quiero llenar un gran archivo `XML` o plantillas, sólo un script. Recuerda, también he descargado tu framework rival. Ya sabes, ¿el que siempre clama ser mucho mejor que el tuyo en los foros? Si todo está trabajando, al tutorial.

Hay un tutorial, ¿no? ¿Uno que me habla en un lenguaje que pueda entender?

Y si el tutorial menciona mi problema, me animaré. Ahora estoy leyendo sobre las cosas que puede hacer para que comience a ponerse interesante, incluso divertido. Me reclinaré y tomaré mi té –¿mencioné que soy del Reino Unido?– y jugaré con tus ejemplos y aprenderé a usar tu creación. Si resuelve mi problema, te enviaré un correo de agradecimiento. Enviaré reportes de error cuando colapse y sugerencias de características

también. Incluso le diré a todos mis amigos que es mejor tu software, aunque nunca probé el de tu rival. Y todo porque cuidaste mis primeros pasos tentativos.

¿Cómo pude haber dudado de ti?

La comunicación entre procesos afecta el tiempo de respuesta de la aplicación

Por Randy Stafford · Traducción: Espartaco Palma

El tiempo de respuesta es crítico en la usabilidad del software. Pocas cosas son tan frustrantes como esperar a que responda algún sistema de software, especialmente cuando nuestra interacción involucra ciclos repetidos de estímulos y respuestas. Nos sentimos como si el software estuviera desperdiciando nuestro tiempo y afectando nuestra productividad. Sin embargo, las causas del pobre tiempo de respuesta son poco apreciadas, especialmente en las aplicaciones modernas. Mucha literatura de administración de rendimiento aún se enfoca en estructuras de datos y algoritmos, temas que pueden hacer una diferencia en algunos casos, pero que son mucho menos propensos a dominar el rendimiento en las modernas aplicaciones empresariales multicapa.

Cuando el rendimiento es un problema en tales aplicaciones, mi experiencia ha sido que examinar estructuras de datos y algoritmos no es el lugar adecuado para buscar mejoras. Los tiempo de respuesta dependen más del número de comunicaciones remotas entre procesos (IPC, ***inter- process communications***) conducidas en respuesta a un estímulo. Aunque puede haber otros cuellos de botella locales, el número de IPC remotas domina usualmente. Cada IPC remota contribuye a latencia no-despreciable para el tiempo de respuesta global, y estas contribuciones remotas se suman, especialmente cuando incurren en secuencia.

Un buen ejemplo es la carga ondular en una aplicación usando mapeo objeto-relación (ORM, ***object-relational mapping***). La carga ondular describe la ejecución secuencial de muchas llamadas a la base de datos para seleccionar los datos necesarios para construir un objeto gráfico (vea: “Lazy Load”, del libro de Martin Fowler: Patterns of Enterprise Application Architecture). Cuando el cliente de la base de datos es un servidor de aplicaciones de capa intermedia renderizando una página web, estas llamadas a la base de datos son ejecutadas usualmente en secuencia en un solo hilo. Sus latencias individuales se acumulan, lo que contribuye al tiempo de respuesta global. Incluso si cada llamada a la base de datos toma sólo 10 minutos, una página que requiera mil llamadas (que no es poco común) exhibirá al menos un tiempo de respuesta de 10 segundos. Otros ejemplos incluyen la invocación de servicios web, respuestas HTTP desde un navegador web, invocación de objetos distribuidos, mensajería de petición-respuesta (reply-request) e interacción con

redes de datos en protocolos de red personalizados. Entre más IPC remotas se necesiten para responder a un estímulo, mayor será el tiempo de respuesta.

Hay algunas estrategias relativamente obvias y bien conocidas para reducir el número de IPC remotas por estímulo. Una estrategia es aplicar el principio de parsimonia, optimizando la interfaz entre procesos, así el número exacto de datos para el propósito a mano es intercambiado con la mínima cantidad de interacciones. Otra estrategia es paralelizar las IPC donde sea posible, así el tiempo de respuesta global es llevado principalmente por la latencia de IPC más larga. Una tercera estrategia es almacenar en caché los resultados de IPC previos, así los futuros IPC pueden ser evitados al usar caché local en su lugar.

Cuando estés diseñando una aplicación, ten en cuenta el número de IPC en respuesta a cada estímulo. Al analizar aplicaciones que sufren de un rendimiento pobre, a menudo me he encontrado radios de IPC-estímulo de miles a 1. La reducción de este radio, ya sea mediante caché, paralelizando o alguna otra técnica, vale mucho más la pena que cambiar la selección de estructuras de datos o ajustar un algoritmo de ordenamiento.

Mantén limpia la compilación

Por Johannes Brodwall · Traducción: Espartaco Palma

¿Alguna vez has visto una lista de advertencias de compilación del largo de un ensayo sobre mala codificación y pensado: “debería hacer algo al respecto, pero ahora no tengo tiempo”? Por otro lado, ¿alguna vez has visto esa solitaria advertencia que acaba de aparecer en una compilación y simplemente la arreglaste?

Cuando inicio un nuevo proyecto desde cero no hay advertencias, no hay desorden, no hay problemas. Pero conforme crece la base de código, si no pongo atención, el desorden, las costras, las advertencias y los problemas pueden empezar a apilarse. Cuando hay mucho ruido, es más difícil encontrar la advertencia que realmente quiero leer entre los cientos de advertencias que no me importan.

Para hacer las advertencias útiles de nuevo, trato de usar una política de tolerancia cero a advertencias desde la compilación. Incluso si la advertencia no es importante, le hago frente. Si no es crítica, pero aún relevante, la arreglo. Si el compilador advierte sobre una potencial excepción de puntero nulo, arreglo la causa, incluso si “sé” que el problema nunca se presentará en producción. Si la documentación embebida (Javadoc o similar) hace referencia a parámetros que han sido quitados o renombrados, limpio la documentación.

Si es algo que realmente no me importa, pregunto al equipo si podemos cambiar nuestra política de advertencias. Por ejemplo, encontré que documentando los parámetros y un valor de retorno de un método en muchos casos no agrega ningún valor, así que no debería ser una advertencia si faltan. O al actualizar una nueva versión del lenguaje de programación el código que anteriormente estaba bien ahora emita advertencias. Por ejemplo, cuando Java 5 introdujo `generics` todo el código antiguo que no especificaba el parámetro de tipo generic nos daba una advertencia. Éste es el tipo de advertencias por las que no quiero ser molestado (al menos, todavía no). Tener un conjunto de advertencias que está fuera del camino de la realidad no le sirve a nadie.

Al asegurarme de que la compilación está siempre limpia no tendré que decidir si una advertencia es irrelevante cada vez que me la encuentro. Ignorar cosas es un trabajo mental y necesito deshacerme de todo el trabajo mental innecesario que pueda. Tener una compilación limpia también hace fácil para alguien más hacerse cargo de mi trabajo. Si dejo las advertencias, alguien más tendrá que encontrar qué es relevante y qué no lo es. O simplemente ignorar todas las advertencias, incluyendo las importantes.

Las advertencias de tu compilador son útiles. Sólo necesitas deshacerte del ruido para empezar a notarlas. No esperes hacer esa “gran limpieza”. Cuando alguna aparece y no la quieres ver, hazle frente de inmediato. También corrige la fuente de la advertencia, suprime esa advertencia o corrige las políticas de advertencia de tu herramienta. Mantener limpia la compilación no se trata sólo de mantenerla limpia de errores de

compilación o fallos de pruebas: las advertencias son también una parte importante y fundamental de la higiene del código.

Aprende a usar las herramientas de línea de comandos

Por Carroll Robinson · Traducción: Espartaco Palma

Hoy en día, muchas herramientas de desarrollo de software se empaquetan como Entornos Integrados de Desarrollo (`IDE`, Integrated Development Environments). Microsoft Visual Studio y el proyecto de software libre Eclipse son dos ejemplos populares, aunque hay muchos otros. Hay muchas razones por las cuales nos gustan los IDE. No sólo porque son fáciles de usar, sino que también alivian al programador de pensar en un montón de pequeños detalles que involucran el proceso de construcción.

La facilidad de uso, sin embargo, tiene su lado negativo. Por lo general, cuando una herramienta es fácil de usar, es debido a que está tomando decisiones por ti y haciendo un montón de cosas automáticamente detrás de la escena. Por lo tanto, si un IDE es el único entorno de programación que siempre has usado, quizás nunca entiendas completamente lo que tus herramientas están haciendo. Haces clic en un botón, algo de magia ocurre, y un archivo ejecutable aparece en la carpeta del proyecto.

Al trabajar con las herramientas de línea de comandos vas a aprender mucho más sobre lo que están haciendo cuando se está construyendo el proyecto. Escribir tus propios archivos `make` te ayudará al entendimiento de todos los pasos (compilar, ensamblar, enlazar, etcétera) que están en la construcción de un archivo ejecutable. Experimentar con las muchas opciones de la línea de comandos de esas herramientas también es una experiencia educacional valiosa. Para empezar con el uso de las herramientas de construcción en línea de comandos, puedes usar las de software libre, como GCC, o las proporcionadas por tu IDE propietario. Después de todo, un IDE bien diseñado es sólo una interface gráfica para un conjunto de herramientas de línea de comandos.

Además de mejorar tu entendimiento del proceso de construcción, hay algunas tareas que pueden ser realizadas de forma más fácil o eficiente con las herramientas de línea de comandos que con un IDE. Por ejemplo, las capacidades de buscar y reemplazar provistas por las utilerías `grep` y `sed` son más poderosas que aquellas que encuentras en IDEs. Las herramientas de línea de comandos inherentemente soportan secuencias de comandos (scripting), lo cuál permite la automatización de tareas, como calendarizar ***builds*** diarios, crear múltiples versiones de un proyecto y la ejecución de conjuntos de pruebas. En un IDE este tipo de automatización puede ser más difícil (si no imposible) de realizar debido a que las opciones de construcción son usualmente especificadas usando cajas de diálogo del `GUI` (Interface Gráfica de Usuario) y el proceso de construcción es invocado con el clic del ratón. Si nunca has dado un paso fuera de un IDE, quizá nunca te diste cuenta de que estos tipos de tareas automatizadas son posibles.

Pero, espera. ¿Acaso el IDE no existe para hacer el desarrollo más fácil y para mejorar la productividad del programador? Bueno, sí. La propuesta presentada aquí no es que debes dejar de usar un IDE. La propuesta es que deberías “mirar debajo de la cortina” y entender lo que el IDE está haciendo por ti. La mejor manera de hacerlo es aprender a usar las herramienta de línea de comandos. Luego, cuando vuelvas a usar tu IDE, tendrás un mucho mejor entendimiento de qué es lo que está haciendo por ti y cómo puedes controlar el proceso de construcción. Por otra parte, una vez que domines el uso de las herramientas de línea de comandos y experimentes el poder y flexibilidad que ofrecen, quizás podrías encontrar que prefieres la línea de comando sobre el IDE.

CONSEJO 44 DE 97

Conoce bien más de dos lenguajes de programación

Por Russel Winder · Traducción: Espartaco Palma

La psicología de la gente programadora ha sabido, desde hace mucho tiempo, que la experiencia de programación está relacionada directamente con el número de diferentes paradigmas de programación con que el programador se sienta cómodo. No se refiere sólo saber o saber un poco, sino a poder programar genuinamente con ellos.

Cada programador inicia con un lenguaje de programación. Este lenguaje tiene un efecto dominante en la forma en que el programador piensa acerca del software. No importa cuántos años de experiencia tenga el programador usándolo, si se queda con él sólo sabrá ese lenguaje. Un programador de un solo lenguaje está limitado en su pensamiento por ese lenguaje.

Un programador que aprende un segundo lenguaje será desafiante, especialmente si tiene un modelo computacional diferente que el primero. C, Pascal, Fortran, todos tiene fundamentalmente el mismo modelo computacional. Cambiar de Fortran a C introduce unos pocos, pero no muchos retos. Moverse de C o Fortran a C++ o Ada introduce retos fundamentales en la forma en que los programas se comportan. Pasarse de C++ a Haskell es un cambio significativo y, por lo tanto, un desafío importante. Moverse de C a Prolog es un desafío muy concreto.

Podemos enumerar varios paradigmas de computación: procedimental, orientado a objetos, funcional, lógico, de flujo de datos (dataflow), etc. Moverse entre estos paradigmas crea los mayores desafíos.

¿Por qué son buenos estos desafíos? Tiene que ver con la forma en que pensamos en la implementación de algoritmos, los modismos y patrones de implementación que aplican. En específico, la fertilización cruzada es la base de la experiencia. Los modismos para la solución de problemas que aplican en un lenguaje podrían no ser posibles en otro. Tratar de portar el modismo de un lenguaje a otro nos enseña sobre ambos lenguajes y sobre el problema a ser resuelto.

La fertilización cruzada en el uso de lenguajes de programación tiene enormes efectos. Quizás el más obvio es el uso creciente de modos de expresión declarativos en los sistemas implementados en lenguajes imperativos. Cualquier persona versada en **programación funcional** puede aplicar fácilmente un enfoque declarativo cuando está usando un lenguaje como lo es C. El uso de enfoques declarativos generalmente conduce a programas más cortos y más comprensibles. C++, por ejemplo, sin duda, toma esto en cuenta con su apoyo incondicional de la programación genérica, que casi necesita un modo de expresión declarativo.

La consecuencia de todo esto es que le incumbe a cada programador ser diestro en la programación en, al menos, dos diferentes paradigmas, e idealmente en los cinco arriba mencionados. Los programadores deben estar siempre interesados en aprender nuevos lenguajes, preferiblemente de un paradigma en el que no

están familiarizados. Incluso si en el trabajo diario siempre usa el mismo lenguaje de programación, la mayor sofisticación en el uso de ese lenguaje cuando una persona puede hacer fertilización cruzada desde otro paradigma no debe ser subestimada. Los empleadores deberían tomar esto en cuenta y permitir en su presupuesto de capacitación aprender lenguajes que actualmente no están siendo usados, como un modo de incrementar la sofisticación de los lenguajes que se utilizan.

A pesar de que es un inicio, un curso de capacitación de una semana no es suficiente para aprender un nuevo lenguaje, generalmente toma unos cuantos meses de uso, aunque a tiempo parcial, para ganar un conocimiento adecuando de un lenguaje. Son sus modismos de uso, no sólo la sintaxis y el modelo computacional, los factores importantes.

CONSEJO 45 DE 97

Conoce tu IDE

Por Heinz Kabutz · Traducción: Espartaco Palma

En la década de los ochenta nuestros entornos de programación eran, por lo general, nada mejor que editores de texto glorificados... si teníamos suerte. El resaltado de sintaxis, que damos por sentado hoy en día, era un lujo que ciertamente no estaba disponible para todos. Los ***Pretty Printers*** para formatear bien nuestro código eran usualmente herramientas externas que tenían que ser ejecutadas para corregir nuestro espaciamiento. Los depuradores eran también programas separados ejecutándose paso a paso a través de nuestro código, pero con un montón de teclazos crípticos.

Durante la década de los noventa las compañías comenzaron a reconocer el potencial de ingresos que pudieran derivarse de equipar a los programadores con mejores y más útiles herramientas. El Entorno Integrado de Desarrollo (`IDE`, por sus siglas en inglés) combinaba las características de edición previas con un compilador, un depurador, Pretty Printer y otras herramientas. Durante ese tiempo, los menús y el ratón también se volvieron populares, lo cuál significaba que los desarrolladores ya no necesitaban aprender combinaciones de teclas crípticos para usar sus editores. Podían simplemente seleccionar su comando en el menú.

En el siglo XXI los IDE se convirtieron en un lugar tan común que eran regalados por las compañías que deseaban ganar un segmento del mercado en otras áreas. El IDE moderno está equipado con una increíble variedad de características. Mi favorita es la **refactorización** automatizada, particularmente la ***Extracción de Método***, en el cual puedo seleccionar y convertir un fragmento de código en un método. La herramienta de refactorización recogerá todos los parámetros que deben ser transferidos al método, lo cuál hace extremadamente fácil modificar código. Mi IDE detectará incluso otro fragmento de código que podría también ser reemplazado por este método y preguntarme si deseo reemplazarlo también.

Otra característica sorprendente en los IDE modernos es la capacidad de hacer cumplir las reglas de estilo dentro de una empresa. Por ejemplo, en Java, algunos programadores han empezado a hacer todos los parámetros como `final` (lo cual, en mi opinión, es una pérdida de tiempo). Sin embargo, como ellos lo tienen como una regla de estilo, todo lo que necesitaría hacer a continuación es configurarlo en mi IDE: obtendría algunas advertencias por cada parámetro que no fuese `final`. Las reglas de estilo también pueden ser utilizadas para encontrar errores probables, tales como comparar objetos autoboxed para la igualdad de referencia, por ejemplo, usando == en los valores primitivos que están autoboxed en referencias a objetos.

Desafortunadamente, los IDE modernos no requieren de invertir esfuerzo para aprender a usarlos. Cuando programé por primera vez en C bajo Unix tuve que pasar un poco de tiempo aprendiendo cómo trabajaba el editor vi, debido a su curva de aprendizaje. Este tiempo gastado pagó por adelantando bellamente al paso de los años. Incluso he escrito el borrador de este artículo con vi. Los IDE modernos tienen una curva de

aprendizaje muy gradual, la cual puede tener como consecuencia que nunca progresamos más allá del uso básico de la herramienta.

Mis primeros pasos al aprender un IDE es memorizar los atajos de teclado. Ya a que mis dedos están en el teclado cuando estoy escribiendo mi código, presionar ***Ctrl+Shift+I*** para alinear una variable me ahorra tener que romper mi flujo, navegar por el menú con el ratón interrumpe este flujo. Estas interrupciones lleva a cambios de contexto innecesarios, haciéndome mucho menos productivo si trato de hacer todo por el camino perezoso. La misma regla también aplica a las habilidades del teclado: aprende a teclear, no te arrepentirás del tiempo invertido por adelantado.

Por último, como programadores tenemos herramientas de flujo Unix que pueden ayudarnos a manipular el código. Como si durante una revisión de código me doy cuenta de que los programadores han nombrado muchas de sus clases de la misma forma, puedo encontrarlas fácilmente usando las herramientas find, sed, sort, uniq y grep, por ejemplo:

EJEMPLO DE CÓDIGO

```
find . -name "*.java" | sed 's/.*\///' | sort | uniq -c | grep -v "^ *1 " | s
```

Esperamos que un plomero que llega a nuestra casa sea capaz de usar su soplete. Pasemos un poco de tiempo estudiando cómo ser más efectivos con nuestro IDE.

Conoce tus límites

Por Greg Colvin · Traducción: Espartaco Palma

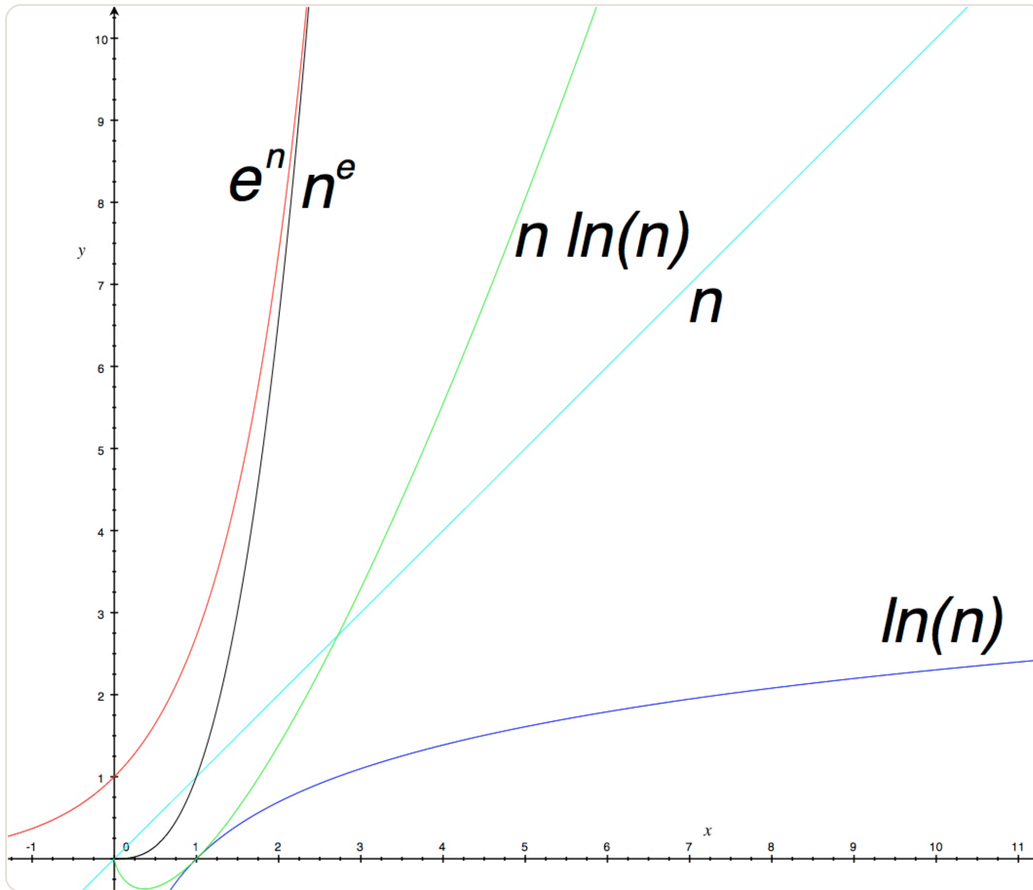
IDEA CLAVE

"Man's got to know his limitations." — Dirty Harry

Tus recursos son limitados. Sólo tienes cierto tiempo y dinero para hacer tu trabajo, incluyendo el tiempo y dinero necesario para mantener al día tus conocimiento, habilidades y herramientas. Sólo se puede trabajar duro, rápido e inteligentemente por cierto tiempo. Tus herramientas son poderosas. Tus máquinas destino son poderosas. Tienes que respetar los límites de tus recursos.

¿Cómo respetar estos límites? Conócete a ti mismo, conoce a tu gente, tu presupuesto y tus cosas. Especialmente, como ingeniero de software, conoce el espacio y tiempo de la complejidad de tus estructuras de datos y algoritmos, así como las características y rendimiento de tus sistemas. Tu trabajo es crear el enlace óptimo de software y sistemas.

La complejidad del espacio y tiempo están dadas como la función $O(f(n))$ donde n es igual al tamaño de las entradas en el espacio asintótico o el tiempo requerido conforme n incrementa hacia infinito. Las clases de complejidad importantes para $f(n)$ incluyen $\ln(n)$, n, $n \ln(n)$, n^e y e^n. Al graficar estas funciones se muestra claramente cómo conforme n se incrementa, $O(\ln(n))$ es siempre mucho más pequeña que $O(n)$ y $O(n \ln(n))$, las cuales son cada vez más pequeñas que $O(ne)$ y O(en). Como decía Sean Parent, para lograr n todas las clases de complejidad se acumulan casi constantemente, casi lineal o casi al infinito.



El análisis de complejidad está en términos de una máquina abstracta, pero el software se ejecuta en máquinas reales. Las sistemas modernos de computadoras están organizados como jerarquías de máquinas físicas y virtuales, incluyendo lenguajes en tiempo de ejecución, sistemas operativos, CPU, memoria caché, memoria de acceso aleatorio, manejadores de disco y redes. La primera tabla muestra los límites en el tiempo de acceso aleatorio y la capacidad de almacenamiento para un servidor en red típico.

	Tiempo de Acceso	**Capacidad**
register	< 1 ms	64b
cache line		64B
L1 cache	1 ms	64 KB
L2 cache	4 ns	8 MB
RAM	20 ns	32 GB
disk	10 ms	10 TB
LAN	20 ms	> 1 PB
internet	100 ms	> 1 ZB

Toma en cuenta que la capacidad y velocidad difiere en varios órdenes de magnitud. El almacenamiento en caché y el ***lookahead*** son usados ampliamente en cada nivel de nuestro sistema para ocultar esta variación,

pero sólo funcionan cuando el acceso es predecible. Cuando el caché falla es frecuente que el sistema esté arrastrándose. Por ejemplo, inspeccionar aleatoriamente cada ***byte*** en un disco duro podría tomar hasta 32 años. Incluso inspeccionar aleatoriamente cada byte en la RAM podría tomar 11 minutos. El acceso aleatorio no es predecible. ¿Qué lo es? Eso depende del sistema, pero volver a acceder a elementos recientemente usados y acceder a elementos secuencialmente suele ser una victoria.

Los algoritmos y las estructuras de datos varían en qué tan efectivamente usan el caché. Por ejemplo:

La búsqueda lineal hace buen uso del ***lookahead***, pero requiere ***O(n)*** comparaciones.

La búsqueda binaria de una matriz ordenada requiere sólo ***O(log(n))*** comparaciones.

La búsqueda en un árbol ***van Emde Boas*** es ***O(log(n))*** y es ajeno al caché.

¿Cuál elegir? Como en el pasado análisis, midiéndolo. La segunda tabla muestra el tiempo requerido para buscar en matrices de enteros de 64 bits vía estos tres métodos. En mi computadora:

La búsqueda lineal es competitiva para matrices pequeñas, pero pierde exponencialmente para matrices grandes

van Emde Boas gana sin usar las manos, gracias a su patrones de acceso predecible.

Elementos	lineal	binario	vEB
8	50	90	40
64	180	150	70
512	1200	230	100
4096	17000	320	160

“Pagas tu dinero y te llevas tu elección”. — Punch

Conoce tu próximo Commit

Por Dan Bergh Johnsson · Traducción: Espartaco Palma

Toqué a tres programadores en su hombro y les pregunté qué estaban haciendo. “Estoy refactorizando este método”, respondió el primero. “Estoy agregando algunos parámetros a esta actividad web”, respondió el segundo. El tercero respondió: “estoy trabajando en esta historia de usuario”.

Podría ser que los primeros dos estaban dedicados en el detalle de su trabajo mientras el tercero estaba viendo la escena completa y tenía un mejor enfoque. Sin embargo, cuando pregunté sobre cuándo y a qué le harían `commit`, la escena cambió dramáticamente. Los primeros dos estaban bastante claros sobre qué archivos estarían involucrados y que esto estaría terminado en una hora o dos. El tercer programador respondió: “oh, creo que estaré listo en unos días. Probablemente agregaré unas cuantas clases y quizás cambie de algún modo estos servicios”.

Los primeros dos no carecían de una visión de la meta general. Habían seleccionado tareas que pensaron que los llevaría a una dirección productiva, y que podrían terminar en un par de horas. Una vez que hubieran terminado con esas tareas, seleccionarían una nueva característica o **refactorización** en la cual trabajar. Todo el código escrito era, por lo tanto, realizado con un propósito claro y limitado, con una meta realizable en mente.

El tercer programador no había sido capaz de descomponer el problema y había trabajado en todos los aspectos al mismo tiempo. No tenía idea de cuánto le tomaría, básicamente estaba haciendo programación especulativa, esperando llegar a algún punto donde podría ser posible hacer un commit. Probablemente, el código escrito al inicio de su larga sesión fue pobremente igualado a la solución que salió al final.

¿Qué harían los primeros dos programadores si sus tareas tomaran más de dos horas? Después de darse cuenta de que habían tardado mucho, lo más probable es que desecharan sus cambios, definieran tareas más pequeñas y volvieran a empezar. El mantenerse trabajando sería una carencia de concentración y llevaría el código especulativo al repositorio. Los cambios serían desechados, pero mantendrían su visión.

El tercer programador podría seguir adivinando y tratando desesperadamente de realizar sus parches dentro de algo a lo que pudiera hacerle commit. Después de todo, no puedes tirar los cambios de código que has hecho –eso sería trabajo perdido, ¿no es así?–. Desafortunadamente, no desechar el código lleva a un código un poco extraño que carece de un propósito claro al entrar al repositorio.

En algún momento, incluso los programadores enfocados en el commit, podrían no encontrar algo útil que pueda ser terminado en dos horas. Entonces, irían directamente al modo especulativo, jugueteando con el código y, por supuesto, desechando los cambios en el momento en que alguna idea los lleve a ese camino. Incluso esas sesiones de hacking aparentemente no estructuradas tienen un propósito: aprender sobre el código y ser capaces de definir una tarea que sería constitutiva de un paso productivo.

Conoce tu próximo commit. Si no puedes terminar, tira tus cambios, define una nueva tarea en la que creas, con las ideas que has ganado. Haz experimentación especulativa donde sea necesario, pero no caigas en el modo especulativo sin darte cuenta. No mandes commit de conjeturas a tu repositorio.

Los grandes datos interconectados pertenecen a una base de datos

Por Diomidis Spinellis · Traducción: Espartaco Palma

Si tu aplicación está manejando un conjunto de elementos de datos grandes, persistentes e interconectados, no dudes en almacenarlos en una base de datos relacional. En el pasado los Sistemas de Administración de Bases de Datos Relacionales (RDBMS, por sus siglas en inglés) solían ser caros, escasos, complejos y unas bestias indomables. Ya no es el caso. Hoy en día los RDBMS son fáciles de encontrar, lo más probable es que el sistema que estás usando ya tenga uno o dos instalados. Algunos RDBMS muy capaces, como MySQL y PostgreSQL, están disponibles como software libre, por lo que el costo de compra ya no es un tema. Aún mejor, los llamados sistemas de bases de datos embebidos se pueden vincular como bibliotecas directamente en tu aplicación, requiriendo casi ninguna configuración o administración; dos notables proyectos de software libre son SQLite y HSQLDB. Estos sistemas son extremadamente eficientes.

Si los datos de tu aplicación son más grandes que la RAM del sistema, una tabla indexada del RBDMS tendrá un rendimiento de órdenes de magnitud más rápida que la colección de mapas de tu biblioteca, que gastará páginas de memoria virtual. Los productos de bases de datos modernos pueden crecer fácilmente con tus necesidades. Con el cuidado adecuado, puedes ampliar una base de datos embebida a un sistema más grande cuando sea requerido. Después, puedes cambiar de un producto de software libre a uno mejor soportado o un sistema propietario más poderoso.

Una vez que sepas los trucos de `SQL`, la creación de aplicaciones centradas en bases de datos es una alegría. Después de que hayas almacenado tus datos correctamente normalizados en la base de datos es fácil extraer eficientemente los hechos con una consulta SQL legible; no hay necesidad de escribir ningún código complejo. Un solo comando SQL puede realizar cambios de datos complejos. Para modificaciones únicas, digamos, un cambio en la forma en que organizas los datos, ni siquiera necesitas escribir código: sólo lanza la interface directa de SQL. Esta misma interface también te permite experimentar con consultas, dejando a un lado el ciclo de compilación-edición de un lenguaje de programación regular.

Otra de las ventajas de basar tu código en un RDBMS implica manejar las relaciones entre los elementos de tus datos. Puedes describir las limitaciones de consistencia en los datos en una manera declarativa, evitando el riesgo de que los apuntadores se cuelguen si olvidas actualizar los datos en un caso extremo. Por ejemplo, puedes especificar que en caso de que un usuario sea eliminado, entonces los mensajes enviados por ese usuario deberían ser eliminados también.

También puedes crear enlaces eficientes entre tus entidades almacenadas en la base de datos en el momento que lo desees, simplemente creando un índice. No hay necesidad de realizar caras y extensas refactorizaciones de campos de clases. Además, codificar en torno a una base de datos permite que varias aplicaciones accedan a tus datos en manera segura. Esto hace fácil actualizar tu aplicación para el uso concurrente y también permite codificar cada parte de tu aplicación usando el lenguaje y plataforma más adecuada. Por ejemplo, puedes escribir el ***back-end*** `XML` de una aplicación web en Java, algunos scripts de autoría en Ruby y una interfaz de visualización en Processing.

Finalmente, recuerda que el RDBMS sudará duro para optimizar los comandos SQL, lo que te permitirá concentrarte en la funcionalidad de tu aplicación en vez de la refinación de algoritmos. Los sistemas avanzados de bases de datos incluso tomarán ventaja de los procesadores multi-core a tus espaldas. Y, conforme la tecnología mejora, también el rendimiento de tu aplicación

Aprende un lenguaje extranjero

Por Klaus Marquardt · Traducción: Espartaco Palma

Los programadores necesitamos comunicarnos. Mucho.

Hay periodos en la vida de un programador cuando mucha de su comunicación parece ser con la computadora. Más precisamente, con los programas ejecutándose en esa computadora. Esta comunicación es con respecto a expresar ideas en una forma leíble por la máquina. Sigue siendo un prospecto emocionante: los programas son ideas convertidas en realidad, con virtualmente ninguna sustancia física involucrada.

Los programadores deben tener fluidez en el lenguaje de la máquina, ya sea real o virtual, y en las abstracciones que pueden estar relacionadas con el lenguaje vía herramientas de desarrollo. Es importante aprender muchas abstracciones diferentes, de otro modo algunas ideas se vuelven increíblemente difíciles de expresar. Los buenos programadores necesitan ser capaces de pararse fuera de su rutina diaria, de estar al tanto de otros lenguajes que son expresivos para otros propósitos. La hora siempre llega cuando éste vale la pena.

Más allá de la comunicación con las máquinas, los programadores necesitan comunicarse con sus pares. Los grandes proyectos de hoy en día son más emprendimientos sociales que simplemente una aplicación en el arte de la programación. Es importante entender y expresar más de lo que pueden las abstracciones de máquina. La mayoría de los mejores programadores que conozco es muy fluida en su lengua madre y, por lo general, en otros idiomas también. Esto no es sólo sobre la comunicación con otros: hablar bien un lenguaje nos lleva a una claridad de pensamiento que es indispensable cuando se abstrae un problema. Y también de eso se trata la programación.

Más allá de la comunicación con las máquinas, con uno mismo y con los compañeros, un proyecto tiene muchos stakeholders, la mayoría con una formación diferente o no técnica. Ellos viven en las áreas de pruebas, calidad y despliegue, en mercadeo y ventas, son usuarios finales en alguna oficina (o tienda o casa). Necesitas entenderlos y a sus preocupaciones. Esto es casi imposible si no puedes hablar su lenguaje en su mundo, su dominio. Mientras puedes pensar que una conversación con ellos salió bien, ellos probablemente no.

Si puedes hablar con contadores, necesitas un conocimiento básico de contabilidad, de centros, de costos o capital invertido, capital empleado, et al. Si vas a hablar con mercadólogos o abogados, algo de su jerga y lenguaje (y, por lo tanto, su mente) debería serte familiar. Todos estos lenguajes específicos del dominio necesitan ser dominados por alguien en el proyecto; de preferencia los programadores, ya que son los responsables de llevar las ideas a la vida a través de una computadora.

Y, por supuesto, la vida es más que proyectos de software. Como lo nota Charlemagne, el conocer otro lenguaje es tener otra alma. Para tus contactos más allá de la industria del software serás más apreciado al conocer lenguajes extranjeros. Para saber cuándo escucharlos en vez de hablar. Para saber que la mayor parte del lenguaje es sin palabras.

IDEA CLAVE

“De lo que no se puede hablar, hay que callar”. Ludwig Wittgenstein.

CONSEJO 50 DE 97

Aprende a hacer estimaciones

Por Giovanni Asproni · Traducción: Espartaco Palma

Como programador debes ser capaz de proporcionar estimaciones a tus directivos, colegas y usuarios de las tareas que necesitas realizar, así ellos tendrán una idea razonablemente precisa del tiempo, costo, tecnología y otros recursos necesarios para lograr sus objetivos.

Para poder estimar bien es obvia la importancia aprender algunas técnicas de estimación. En primer lugar, sin embargo, es fundamental aprender qué son las estimaciones y para qué deberían ser usadas –por extraño que parezca, muchos desarrolladores y administradores no conocen esto–.

El siguiente diálogo entre un administrador de proyectos y un programador es nada atípico:

> Administrador de Proyecto: ¿Puedes darme un estimado del tiempo necesario para desarrollar la característica xyz?
>
> Programador: Un mes.
>
> Administrador de Proyecto: ¡Eso es mucho tiempo! Sólo tenemos una semana.
>
> Programador: Necesito al menos tres.
>
> Administrador de Proyecto: Puedo darte dos cuando mucho.
>
> Programador: ¡Es un trato!

Al programador, al final, se le ocurre un "estimado" que concuerda con lo que es aceptable para el administrador. Pero, ya que es una estimación del programador, el gerente lo hará responsable de ello. Para entender qué está mal en esta conversación necesitamos tres definiciones: estimado, fin y compromiso.

> Un estimado es un cálculo aproximado o un juicio de valor, número, cantidad o extensión de algo. Esta definición implica que un estimado es una medición factual basada en datos concretos y experiencia previa; la esperanza y los deseos deben ser ignorados cuando se calcula. La definición también implica que, al ser aproximada, una estimación no pueden ser precisa, por ejemplo: una tarea de desarrollo no puede ser estimada para durar 234.14 días.
>
> Un fin es una declaración de un objetivo deseable del negocio, por ejemplo, "el sistema debe soportar al menos 400 usuarios concurrentes".
>
> Un compromiso es una promesa de ofrecer una funcionalidad especificada a una determinado nivel de calidad en una cierta fecha o evento. Un ejemplo podría ser: "la funcionalidad de búsqueda estará disponible en la próxima versión del producto".

Los estimados, fines y compromisos son independientes uno del otro, pero los blancos y cometidos deberían estar basados en estimados. Como Steve McConnell señala: "El propósito principal de la estimación de

software no es predecir el futuro del proyecto, sino determinar si los fines son lo suficientemente realistas para que pueda ser controlado hasta lograrlo". Por lo tanto, el propósito de una estimación es hacer una administración de proyecto adecuada y una planificación posible, permitiendo que los interesados hagan compromisos basados en fines realistas.

Lo que estaba pidiendo el administrador en la conversación anterior al programador era hacer un compromiso basado en un fin no declarado que el administrador tenía en mente, no dar un estimado. La próxima vez que te pidan proporcionar un estimado asegúrate que todos los involucrados sepan de lo que están hablando, y tus proyectos tendrán una mejor oportunidad de éxito. Ahora es el momento de aprender algunas técnicas...

Aprende a decir "Hola, Mundo"

Por Thomas Guest · Traducción: Espartaco Palma

Paul Lee, nombre de usuario “leep”, comúnmente conocido como Hoppy, tenía la reputación de experto local en temas de programación. Necesitaba ayuda. Caminé hacia el escritorio de Hoppy y le pregunté:

— ¿Podrías echar un vistazo al código por mí?

— Seguro —dijo Hoppy—, toma una silla.

Tuve el cuidado de no derribar las latas vacías de soda apiladas en una pirámide detrás de él.

—¿Qué código?

—En una función en un archivo —le dije.

—Echemos un vistazo a esta función.

Hoppy alejó una copia de K&R y deslizó su teclado frente a mí. ¿Dónde está el `IDE`? Aparentemente Hoppy no tenía un IDE ejecutándose, sólo algún editor que yo no podía operar. Tomó de nuevo el teclado. Unos cuantos teclazos después y teníamos el archivo abierto –era un archivo algo grande– y estamos observando la función –era una función algo grande–. Él avanzó unas páginas hacia el bloque condicional que quería cuestionarle.

— ¿Qué haría realmente esta cláusula si x es negativo? —le pregunté—. ¿Sin duda, es un error.

Había estado probando toda la mañana tratando de encontrar una manera de forzar que x fuera negativo, pero la gran función en un gran archivo era parte de un gran proyecto, y el ciclo de recompilar y volver a ejecutar mis experimentos me estaba venciendo. ¿No podría un experto como Hoppy simplemente decirme la respuesta?

Hoppy admitió que estaba seguro. Para mi sorpresa, no buscó en K&R. En vez de ello, copió el bloque de código en un nuevo buffer del editor, lo reindentó y lo envolvió en una función. Un poco más tarde codificó una función `main` y lo cicló, pidiendo al usuario valores de entrada, pasándolos a la función e imprimiendo el resultado. Guardó el buffer como un nuevo archivo, `tryit.c`. Todo esto lo podría haber hecho yo mismo, creo que quizá no tan rápido. Sin embargo, su siguiente paso fue maravillosamente simple y, para ese tiempo, un poco extraño para mi manera de trabajar

EJEMPLO DE CÓDIGO

```
$ cc tryit.c && ./a.out
```

¡Mira! Su programa, concebido unos pocos minutos antes, ahora estaba en marcha y funcionando. Probamos unos cuantos valores y confirmó mis sospechas (¡había tenido razón sobre algo!) y entonces cotejó la sección correspondiente de K&R. Le agradecí a Hoppy y me fui, una vez más, teniendo cuidado de no molestar su pirámide de latas de soda.

De regreso a mi escritorio, cerré mi IDE. Me había hecho tan familiar al trabajo con un gran proyecto con un gran producto que había empezado a pensar qué debía hacer. Una computadora de propósito general puede realizar pequeñas tareas también. Abrí un editor de texto y empecé a escribir.

EJEMPLO DE CÓDIGO

```
#include <stdio.h>

int main() {
    printf("Hello, World\n");
    return 0;
}
```

Deja que tu proyecto hable por sí mismo

Por Daniel Lindner · Traducción: Espartaco Palma

Tu proyecto probablemente tenga un sistema de control de versiones . Quizás está conectado a un servidor de Integración Continua que verifica la correctitud por medio de **pruebas automatizadas**. Eso es genial.

Puedes incluir herramientas para el análisis estático de código en tu servidor de Integración Continua y así recopilar métricas de código. Estas métricas proveen retroalimentación sobre aspectos específicos, así como la evolución en el tiempo. Al instalar métricas de código, siempre habrá una línea roja que no querrás cruzar. Supongamos que inicias con un 20% de cobertura de pruebas y nunca caes por debajo del 15%. La Integración Continua ayuda a mantener un registro de todos estos números, pero todavía tienes que revisarlos regularmente. Imagina que puedes delegar estas tareas al proyecto mismo y confiarle el reportar cuando las cosas se ponen peor.

Necesitas darle a tu proyecto una voz. Esto puede ser realizado por email o mensajería instantánea, informando a los desarrolladores sobre la última caída o mejora en los números. Pero esto es incluso más efectivo de llevar usando un Dispositivo de Retroalimentación Extrema (XFD, por sus siglas en inglés, ***Extreme Feedback Device***).

La idea del XFD es manejar un dispositivo físico como una lámpara, una fuente portátil, un robot de juguete o incluso un lanza cohetes USB, basado en el resultado del análisis automático. Cada vez que tus límites se rompan, el instrumento altera su estado. En el caso de la lámpara, ésta se enciende, brillante y clara. No puedes olvidar el mensaje, incluso si estás cruzando la puerta para irte a casa.

Dependiendo del tipo de dispositivo de retroalimentación extrema, puedes oír la ruptura del compilado, ver las señales rojas de advertencia en tu código, incluso oler tu código. Los instrumentos pueden ser replicados en distintos lugares si trabajas en un equipo distribuido. Puedes colocar un semáforo en la oficina de tu director de proyecto, indicando el estado general de salud. El director del proyecto te lo agradecerá.

Deja que tu creatividad te guíe al escoger el dispositivo apropiado. Si tu cultura es bastante ***geek***, podrías buscar la manera de equipar a la mascota de tu equipo con juguetes de radio control. Si deseas una apariencia más profesional, invierte en lámparas más estilizadas. Busca más inspiración en Internet. Cualquier cosa con un enchufe de alimentación o un control remoto tiene el potencial de ser usado como un dispositivo de retroalimentación extrema.

El dispositivo de retroalimentación extrema actúa como la caja de voz de tu proyecto. El proyecto ahora se encuentra físicamente con los desarrolladores, quejándose o alabando, de acuerdo a las reglas que el equipo haya escogido. Puedes llevar esta personificación más allá aplicando software de síntesis de voz y un par de altavoces. Ahora tu proyecto realmente habla por sí mismo.

CONSEJO 53 DE 97

El linker no es un programa mágico

Por Walter Bright · Traducción: Espartaco Palma

Con una frecuencia depresiva (me sucedió otra vez justo antes de escribir esto), la visión que tienen muchos programadores sobre el proceso de pasar de código fuente a un ejecutable estáticamente enlazado en un lenguaje compilado es:

Editar código fuente

Compilar el código fuente en archivos ***objeto***

Algo mágico sucede

Ejecutar ejecutable

El paso 3 es, por supuesto, el paso de enlazado. ¿Por qué diría una cosa tan atroz? He estado en soporte técnico por décadas, y tengo las siguientes preguntas una y otra vez:

El linker dice "def está definido más de una vez".

El linker dice "abc es un símbolo sin resolver

¿Por qué el ejecutable es tan grande?

Seguido de "¿ahora qué hago?", usualmente mezclado junto con las frases "parece que" y "de alguna manera", y un aura de total desconcierto. Son los "parece que" y "de alguna manera" los que indican que el proceso de enlace es visto como un proceso mágico, presumiblemente entendible sólo por magos y brujos. El proceso de compilado no provoca este tipo de frases, implicando que los programadores generalmente entienden cómo funcionan los compiladores o, al menos, qué hacen.

Un linker es un programa muy estúpido, ordinario y directo. Todo lo que hace es concatenar el código y las secciones de datos de los archivos ***objeto***, conectar las referencias a los símbolos con sus definiciones, empujar símbolos sin resolver fuera de la biblioteca y escribir un ejecutable. Eso es todo. ¡Sin hechizos! ¡Sin magia! Lo tedioso de escribir un linker es usualmente decodificar y generar formatos de archivo ridículamente complicados, pero eso no cambia la esencia natural de un linker.

Digamos que el linker está diciendo "def está definido más de una vez". Muchos lenguajes de programación, tales como C, C++, y D, tienen ambos, declaraciones y definiciones. Las declaraciones normalmente van en archivos de encabezados, tales como:

EJEMPLO DE CÓDIGO

```
extern int iii;
```

lo cual genera una referencia externa al símbolo `iii`. Una definición, por otro lado, en realidad establece el almacenamiento para el símbolo, usualmente aparece en el archivo de implementación, y luce así:

EJEMPLO DE CÓDIGO

```
int iii = 3;
```

¿Cuántas definiciones puede haber por cada símbolo? Como en la película Highlander, sólo puede haber una. Así que, ¿qué tal si una definición de `iii` aparece en más de un archivo de implementación?

EJEMPLO DE CÓDIGO

```
// Archivo a.c
int iii = 3;

// Archivo b.c
double iii(int x) { return 3.7; }
```

El linker se quejará porque `iii` está siendo definido varias veces.

No sólo puede haber uno, debe haber uno. Si `iii` sólo aparece como una declaración, pero nunca en una definición, el linkers se quejará sobre `iii` por ser un símbolo no resuelto.

Para determinar por qué un ejecutable es del tamaño que es, dale un vistazo al archivo de mapa que los linkers generan opcionalmente. Un archivo de mapa es una lista de todos los símbolos en el ejecutable junto con sus direcciones. Te dice qué módulos fueron enlazados desde la biblioteca y el tamaño de cada módulo. Ahora puedes ver de dónde viene tanta hinchazón. Frecuentemente habrá módulos de biblioteca que no tendrás idea por qué fueron enlazados. Para saberlo, quita temporalmente el módulo sospechoso de la biblioteca y vuelve a enlazar. El error de "símbolo no definido" generado indicará quién está referenciando ese módulo.

Aunque no siempre es obvio por qué aparece un mensaje del linker en particular, no hay nada mágico sobre los linkers. La mecánica es directa: son los detalles lo que tienes que averiguar en cada caso.

La longevidad de las soluciones provisionales

Por Klaus Marquardt · Traducción: Espartaco Palma

¿Por qué creamos soluciones provisionales?

Típicamente hay algún problema inmediato que resolver. Puede ser provisional para el equipo de desarrollo, algunas herramientas que llenan un vacío en la cadena de herramientas. Puede ser externo, visible al usuario final, como un solución que aborda la funcionalidad faltante.

En muchos sistemas y equipos encontrarás algún software que está algo desintegrado del sistema, que es considerado un borrador para ser cambiado en algún momento, que no sigue el estándar y las guías que dan forma al resto del código. Inevitablemente oirás a desarrolladores quejándose sobre esto. Las razones para su creación son muchas y variadas, pero la clave para el éxito de una solución provisional es simple: es útil.

Las soluciones interinas, sin embargo, adquieren inercia (o momentum, dependiendo de tu punto de vista) debido a que están ahí, útiles y ampliamente aceptadas, no hay necesidad inmediata para hacer algo más. Sin embargo, cuando la parte interesada tiene que decidir qué acción agrega más valor, habrá muchos cosas que ranqueen más algo que la instalación apropiada de una solución provisional. ¿Por qué? Porque está ahí, funciona y es aceptada. El único lado malo perceptible es que no sigue los estándares seleccionados y directrices elegidas, excepto en un pequeño nicho del mercado, esto no es considerado como una fuerza significativa.

Así que la solución provisional se mantiene en su lugar. Por siempre.

Y si un problema surge con esa solución provisional, es poco probable que se provea una actualización que esté en línea con la calidad de producción aceptable. ¿Qué hacer? Una rápida actualización en esa solución provisional a menudo hace el trabajo. Y será más común que sea bien recibida. Exhibe las mismas fortalezas que la solución provisional inicial… Sólo está más actualizada.

¿Es esto un problema?

La respuesta depende de tu proyecto, y de su interés personal en las normas del código de producción. Cuando los sistemas contienen muchas soluciones provisionales, su entropía o complejidad interna crece y su **mantenibilidad** disminuye. Sin embargo, quizás de inicio nuestra pregunta sea la equivocada. Recuerda que estamos hablando sobre una solución. Podría no ser tu solución preferida –es poco probable que sea la solución preferida de alguien–, pero es débil la motivación para rehacer esta solución.

¿Qué podríamos hacer si vemos un problema??

- Evitar crear una solución provisional en primer lugar.
- Cambiar las fuerzas que influencian la decisión del Administrador de Proyecto.
- Dejarlo como está.

Vamos a examinar estas opciones más de cerca.

> Eludir no funciona en la mayoría de los casos. Hay un problema a resolver y los estándares pasan a ser muy restrictivos. Puedes gastar energía para cambiar los estándares. Una honorable aunque tediosa tarea… y ese cambio no será efectivo a tiempo para el problema actual.
>
> Las fuerzas están arraigadas en la cultura del proyecto, la cuál se resiste a cambios voluntarios. Podría tener éxito en proyectos pequeños –especialmente si sólo eres tú– y acabas de limpiar el desorden sin preguntar antes. También podría tener éxito si el proyecto es tan confuso que se ha estancado visiblemente y tomarse algún tiempo para la limpieza suele ser aceptado.
>
> El estatus quo automáticamente aplica si la opción no lo hace.

Crearás muchas soluciones, algunas serán provisionales, muchas serán útiles. La mejor manera de superar las soluciones provisionales es hacerlas superfluas, proveer una más elegante y útil solución. Podrías recibir la serenidad de aceptar las cosas que no puedes cambiar, coraje para cambiar las cosas que puedes y sabiduría para saber la diferencia.

Haz las Interfaces fáciles de usar correctamente y difíciles de usar incorrectamente

Por Scott Meyers · Traducción: Espartaco Palma

Una de las tareas más comunes en el desarrollo de software es la especificación de la interfaz. Las interfaces ocurren al más alto nivel de abstracción (interfaces de usuario), en la más baja (interfaces de función) y en los niveles intermedios (interfaces de clases, de bibliotecas, etcétera). Independientemente de que estés trabajando con el usuario final para especificar cómo estará interactuando con un sistema, colaborando con desarrolladores para especificar un `API` o declarando funciones privadas para una clase, el diseño de interfaz es una parte importante de tu trabajo. Si lo haces bien, será un placer usar tus interfaces y aumentará la productividad de los demás. Si lo haces pobremente, tus interfaces serán la fuente de frustraciones y errores.

Las buenas interfaces son:

> Fáciles de usar correctamente. La gente que usa una interfaz bien diseñada casi siempre usa la interfaz correctamente, porque es la ruta de menor resistencia. En una Interfaz Gráfica de Usuario (`GUI`) siempre hacen clic en el ícono, botón o entrada de menú correcta, debido a que es obvio y algo fácil de hacer. En una API casi siempre pasan los parámetros correctos con el valor correcto, debido a que es la manera más natural. Con interfaces que son fáciles de usar correctamente, la cosas funcionan.
>
> Difíciles de usar incorrectamente. Las buenas interfaces se anticipan a los errores que la gente comete y hace que sea difícil –idealmente imposible– realizarlos. Una GUI debería deshabilitar o remover comandos que no tengan sentido en el contexto actual, por ejemplo, o una API debería eliminar la secuencia de argumentos al permitir que los parámetros sean pasados en cualquier orden.

Una buena manera de diseñar interfaces que son fáciles de usar correctamente es hacer ejercicios antes de que existan. Simula una GUI –en un pizarrón o usando fichas en una mesa– y juega con ellos antes de que cualquier código haya sido creado. Escribe llamadas a la API antes de que las funciones hayan sido declaradas. Revisa los casos de uso comunes y especifica cómo quieres que se comporten las interfaces. ¿En qué quieres que puedan hacer clic? ¿Qué quieres pasarle? Las interfaces fáciles de usar parecen naturales, debido a que te dejan hacer lo que quieres hacer. Es más frecuente dar con esas interfaces si las

desarrollas desde el punto de vista de los usuarios (esta perspectiva es una de las fortalezas de la programación test-first).

Hacer las interfaces difíciles de usar incorrectamente requiere dos cosas. Primero, debes anticiparte a los errores que los usuarios podrían hacer y encontrar la manera de prevenirlos. Segundo, debes observar cómo una interfaz es usada erróneamente durante las primeras liberaciones y modifica la interfaz –¡sí, modificar la interfaz!– para prevenir tales errores. La mejor manera de prevenir el uso incorrecto es hacer tal uso imposible. Si los usuarios siguen queriendo hacer un “deshacer” en una acción irrevocable, intenta hacer la acción revocable. Si ellos siguen pasando un valor erróneo a la API, mejor modifica la API para tomar los valores que el usuario quiere pasar.

Sobre todo, recuerda que las interfaces existen para la conveniencia de sus usuarios, no la de sus implementadores.

Haz lo invisible más visible

Por Jon Jagger · Traducción: Espartaco Palma

Muchos aspectos de la invisibilidad son correctamente dichos como principios a usar. Nuestra terminología es rica en metáforas de invisibilidad, mecanismos de transparencia y ocultamiento de información, para mencionar sólo dos. El software y el proceso de desarrollo pueden ser, para parafrasear a Douglas Adams, casi invisibles:

> El código fuente no tiene una innata presencia o comportamiento, y no obedece las leyes de la física. Es visible cuando lo cargas en un editor, pero cierra el editor y se ha ido. Piensa sobre eso un rato y, como el árbol cayendo cuando nadie lo escucha, empieza a preguntarte si en realidad existe.
>
> Una aplicación en ejecución tiene presencia y comportamiento, pero no revela nada del código fuente con el que fue construido. La página principal de Google es placenteramente minimalista; lo que pasa detrás es lo realmente sustancial.
>
> Si has terminado el 90% y estás eternamente atorado tratando de debugear el último 10% entonces no has acabado el 90%, ¿o sí? Corregir errores no es progresar. No te pagan por debugear. El debugging es un derroche. Es bueno hacer una pérdida más visible así puedes ver qué es y empezar a pensar en no crearla, en primer lugar.
>
> Si tu proyecto está aparentemente en camino y una semana después está seis meses atrasado, tienes problemas, el más grande de ellos probablemente no sea que estás seis meses tarde, ¡sino que el campo de invisibilidad es lo suficientemente poderoso como para ocultar seis meses de retraso! La falta de progreso visible es sinónimo de la falta de progreso.

La invisibilidad puede ser peligrosa. Piensas más claramente cuando tienes algo concreto a qué amarrar tu pensamiento. Administras mejor las cosas cuando puedes verlas y verlas cambiar constantemente:

> Escribir pruebas unitarias provee evidencia sobre qué tan fácil es el código unitario con respecto a la prueba unitaria. Ayuda a revelar la presencia (o ausencia) de cualidades de desarrollo que te gustaría que el código exhiba; cualidades como bajo acoplamiento y alta cohesión.
>
> Ejecutar pruebas unitarias provee evidencia sobre el comportamiento del código. Ayuda a revelar la presencia (o ausencia) de cualidades en tiempo de ejecución que te gustaría que la aplicación exhiba; cualidades como la fortaleza y la correctitud.
>
> El usar tableros de boletines y tarjetas hace el progreso más visible y concreto. Las tareas pueden ser vistas como “No iniciadas”, “En progreso” o “Terminadas” sin la referencia a una herramienta de administración de proyectos y sin tener que perseguir a los programadores para que entreguen reportes de estatus ficticios.

> Realizar desarrollo incremental aumenta la visibilidad del progreso del desarrollo (o la falta de él) al incrementar la frecuencia de la evidencia del desarrollo. El completar la liberación del software revela realidad; los estimados no.

Es mejor desarrollar software con una gran cantidad de evidencia visible habitual. La visibilidad otorga confianza de que el progreso es genuino y no una ilusión, deliberado y no involuntario, repetible y no accidental.

El paso de mensajes lleva a una mejor escalabilidad en sistemas paralelos

Por Russel Winder · Traducción: Espartaco Palma

A los programadores se les enseña desde el primer momento de sus estudios en computación que la concurrencia –y especialmente el paralelismo, un subconjunto especial de la concurrencia– es difícil, que sólo los mejores pueden tener la esperanzas de *hacerlo bien* y que incluso se equivocan. Siempre hay una gran atención a ***threads***, semáforos, monitores y lo difícil que es obtener el acceso simultáneo a variables para ser seguro en ***threads***.

Es cierto, hay muchos problemas difíciles, y pueden ser muy difíciles de resolver. Pero, ¿cuál es la raíz del problema? Memoria compartida. Casi todos los problemas de concurrencia que la gente tiene una y otra vez se relacionan con el uso de memoria compartida mutable: ***race conditions***, ***deadlocks***, ***livelock***, etcétera. La respuesta parece obvia: ¡renunciar a la concurrencia o abstenerse de la memoria compartida!

Olvidar la concurrencia casi seguramente no es una opción. Las computadoras tienen más y más núcleos de manera casi trimestral, por lo que el aprovechamiento de cierto paralelismo se hace más y más importante. No nos podemos confiar tanto en cada incremento de la velocidad del procesador para mejorar el rendimiento de nuestra aplicación. Obviamente, no mejorar el rendimiento es una opción, pero es poco probable que sea aceptable para los usuarios.

Entonces, ¿podemos evitar la memoria compartida? Definitivamente.

En vez de usar ***threads*** y memoria compartida como nuestro modelo de programación, podemos usar procesos y el paso de mensajes. Los procesos aquí sólo significan un estado protegido e independiente con código ejecutándose, no necesariamente un proceso del sistema operativo. Lenguajes como Erlang (y Occam antes de él) han mostrado que los procesos son un exitoso mecanismo para la programación de sistemas concurrentes y paralelos. Tales sistemas no tienen todo el estrés de sincronización que la memoria compartida y los sistemas de ***multi- thread*** tienen. Más aún, hay un modelo formal –Proceso de Comunicación Secuencial (CSP, por sus siglas en inglés [***Communicating Sequential Processes***])– que puede ser aplicado como parte de la ingeniería de tales sistemas.

Podemos ir más allá e introducir sistemas de flujo de datos como una forma de computación. En un sistema de flujo de datos no hay un flujo de control explícitamente programado. En vez de eso se configura un grafo directo de operadores conectados por rutas de datos y entonces los datos son alimentados al sistema. La evaluación es controlada por la disponibilidad de los datos dentro del sistema. Definitivamente sin problemas de sincronización.

Dicho todo esto, lenguajes como C, C++, Java, Python y Groovy son el principal lenguaje del desarrollo de sistemas y todos ellos son presentados a los programadores como lenguajes para desarrollo de memoria compartida, sistemas de ***multi-thread***. Entonces, ¿qué se puede hacer? La respuesta es utilizar –o, si no existen, crear– bibliotecas y ***frameworks*** que proporcionan modelos de procesos y paso de mensajes, evitando todo el uso de memoria compartida mutable.

Después de todo, no programar con memoria compartida y usar en vez de eso paso de mensajes es probablemente la forma más exitosa de implementar sistemas que aprovechan el paralelismo que es ahora endémico en el hardware de computación. Quizás extrañamente, pero a pesar de que los procesos son anteriores a los ***threads*** como unidad de concurrencia, el futuro parece estar en usar ***threads*** para implementar procesos.

Mensaje al futuro

Por Linda Rising · Traducción: Espartaco Palma

Quizás sea porque la mayoría de ellos son personas inteligentes, pero en todos estos años he enseñado y trabajado codo a codo con programadores, parece que muchos piensan que debido a que los problemas con que estuvieron luchando eran difíciles, entonces las soluciones deben ser difíciles de entender y mantener para todos (quizás incluso para ellos mismos unos cuantos meses después de que el código haya sido escrito

Recuerdo un incidente con Joe, un estudiante en mi clase de estructuras de datos, quien había venido a mostrarme lo que él había escrito.

—¡Te apuesto que no puedes adivinar qué hace! —gritó. —Estás en lo correcto —estuve de acuerdo, sin gastar mucho tiempo en su ejemplo e imaginándome cómo conseguir un importante mensaje de esto—. Estoy segura de que has estado trabajando duro en esto. Me imagino, sin embargo, que no has olvidado nada importante. Dime, Joe, ¿tienes un hermano menor? —Sí. ¡Claro que sí! ¡Phil! Él está en tu clase de introducción. ¡Está aprendiendo a programar también! —anunció Joe orgullosamente. —Eso está muy bien —repliqué—. Me imagino que él pudo leer este código. —¡De ninguna manera! —dijo Joe—. ¡Esto es algo difícil! —Sólo supón —sugerí— que éste es un código de trabajo real y que en unos pocos años Phil será contratado para hacer una actualización de mantenimiento. ¿Qué has hecho con él?

Joe me miró parpadeando.

—Sabemos que Phil es realmente inteligente, ¿verdad? —Joe asintió—. Y odio decirlo, pero ¡soy bastante inteligente también! —Joe sonrió—. Así que si no puedo entender fácilmente lo que has hecho aquí y tu muy inteligente hermano menor probablemente se rompa la cabeza con esto, ¿qué significa eso de lo que has escrito?

Joe miró su código un poco diferentemente, me pareció.

—¿Qué tal esto? —sugerí con mi mejor voz de "soy tu amigable mentor"—. Piensa en cada línea de código que has escrito como un mensaje para alguien en el futuro, alguien que podría ser tu hermano menor. Pretende que estás explicándole a esta persona inteligente cómo resolver el difícil problema. ¿Es esto lo que te gustaría imaginar? Que un programador inteligente en el futuro vea tu código y diga: "¡Wow! ¡Esto es genial! Puedo entender perfectamente qué ha hecho aquí y estoy impresionado, qué elegante, no, espera, qué hermosa pieza de código es ésta. Voy a mostrárselo a los otros muchachos de mi equipo. ¡Ésta es una obra maestra!. Joe, ¿crees que podrías escribir un código que resuelva este difícil problema, pero sea tan bello que cantaría? Sí, igual que una melodía inquietante. Creo que cualquiera que pueda llegar con la muy difícil solución que tienes aquí también podría escribir algo hermoso. Hmmm... me pregunto si debería empezar a calificar la belleza. ¿Tú qué crees, Joe?

Joe tomó su trabajo y me miró, una pequeña sonrisa se asomó en su cara.

—Lo entiendo, prof, me retiro a hacer un mundo mejor para Phil. Gracias.

Oportunidades perdidas del Polimorfismo

Por Kirk Pepperdine · Traducción: Espartaco Palma

El polimorfismo es una de las grandes ideas fundamentales de la Orientación a Objetos (OO). La palabra, tomada del griego, significa muchas (poli) formas (morfos). En el contexto de programación el polimorfismo se refiere a las muchas formas de una clase particular de objetos o métodos. Pero el polimorfismo no es simplemente sobre implementaciones alternativas. Usado con cuidado, crea diminutos contextos de ejecución que nos dejan trabajar sin la necesidad de detallados bloques if-then-else. Estar en un contexto nos permite hacer lo correcto directamente, mientras que estar fuera del contexto nos obliga a reconstruirlo para entonces poder hacer lo correcto. Con el uso cuidadoso de implementaciones alternadas podemos capturar el contexto que nos ayude a producir menos código que sea más leíble. Esto se demuestra mejor con algo de código, como el siguiente (e irreal) carrito de compras:

EJEMPLO DE CÓDIGO

```
public class ShoppingCart {
    private ArrayList<Item> cart = new ArrayList<Item>();
    public void add(Item item) { cart.add(item); }
    public Item takeNext() { return cart.remove(0);  }
    public boolean isEmpty() { return cart.isEmpty(); }
}
```

Digamos que nuestra compra en línea ofrece elementos que pueden ser descargados y elementos que necesitan ser enviados. Vamos a construir otro objeto que soporte estas operaciones:

EJEMPLO DE CÓDIGO

```
public class Shipping {
    public boolean ship(Item item, SurfaceAddress address) { ... }
    public boolean ship(Item item, EMailAddress address { ... }
}
```

Cuando un cliente ha completado la compra, necesitamos enviar los bienes:

EJEMPLO DE CÓDIGO

```
while (!cart.isEmpty()) {
    shipping.ship(cart.takeNext(), ???);
}
```

El parámetro `???` no es algún nuevo operador ***Elvis***, está preguntado si debería enviar correo electrónico o correo normal. El contexto necesario para responder la pregunta ya no existe. Pudimos haber capturado el método de envío en un boleano o en un enum y entonces usar un `if-then- else` para llenar el parámetro faltante. Otra solución sería crear dos clases, en las cuales ambas extiendan `Item`. Llamémosle `DownloableItem` y `SurfaceItem`.Ahora vamos a escribir algo de código. Promoveré `Item` para que sea una interfaz que soporte un único método: `ship`. Para enviar el contenido del carrito haremos una llamada a `item.ship(shipper)`. Ambas clases `DownloadbleItem` y `SurfaceItem` implementarán `ship`.

EJEMPLO DE CÓDIGO

```
public class DownloadableItem implements Item {
    public boolean ship(Shipping shipper) {
        shipper.ship(this, customer.getEmailAddress());
    }
}

public class SurfaceItem implements Item {
    public boolean ship(Shipping shipper) {
        shipper.ship(this, customer.getSurfaceAddress());
    }
}
```

En este ejemplo hemos delegado la responsabilidad de trabajar con `Shipping` en cada `Item`. Debido a que cada item sabe cómo es mejor que sea enviado, este arreglo nos permite estar con él sin la necesidad de un `if-then-else`. El código también demuestra un uso de dos patrones que frecuentemente actúan juntos: ***Command*** y ***Double Dispatch***. El uso efectivo de estos patrones reside en un uso cuidadoso del polimorfismo. Cuando esto suceda habrá una reducción del número de bloques if-then-else en nuestro código.

Si bien hay casos en los que es mucho más práctico utilizar `if-then- else` en vez del polimorfismo, es más frecuente el caso en el cual un estilo de código más polimórfico dará lugar a un **código base** más pequeño, más fácil de leer y menos frágil. El número de oportunidades perdidas es un simple conteo de declaraciones `if-then-else` en nuestro código.

Noticias raras - Los testers son tus amigos

Por Burk Hufnagel · Traducción: Espartaco Palma

Ya sea que se llamen ellos mismos Aseguramiento de Calidad (QC, ***Quality Check***) o Control de Calidad, muchos programadores los llaman problemas. En mi experiencia, los programadores tienen frecuentemente una relación de confrontación con la gente que prueba su software. "Son demasiado exigentes" y "quieren todo perfecto" son las quejas comunes. ¿Te suena familiar?

No estoy seguro del porqué, pero siempre he tenido una visión diferente de los ***testers***. Quizás es porque el "tester" en mi primer trabajo era la secretaria de la empresa. Margaret era una señora muy agradable que mantenía la oficina funcionando e intentaba enseñar a un par de jóvenes programadores cómo comportarse profesionalmente frente a los clientes. Ella también tenía el don de encontrar cualquier error, no importa lo oscuro, en cuestión de minutos.

En ese entonces estaba trabajando en un programa escrito por un contador que pensaba que era un programador. No es necesario decirlo, tenía algunos problemas serios. Cuando pensaba que tenía una pieza sólida, Margaret intentaría usarlo y, más frecuentemente que nunca, fallaría en alguna forma justo después de algunos teclazos. A veces era frustrante y embarazoso, pero ella era una persona agradable a quien nunca pensé en culpar por hacerme ver mal. Eventualmente llegó el día cuando Margaret fue capaz de iniciar limpiamente el programa, introducir una factura, imprimirla y cerrarlo. Estaba muy emocionado. Aún mejor, cuando lo instalamos en una de las computadoras de nuestros clientes, todo funcionaba. Ellos nunca vieron ningún problema porque Margaret me había ayudado a encontrarlos y arreglarlos primero.

Es por eso que digo que los ***testers*** son tus amigos. Puedes pensar que te hacen ver mal al reportar cuestiones triviales. Pero cuando los clientes están emocionados por no ser molestados con todas esas "pequeñas cosas" que QC te hizo corregir, entonces te verás bien. ¿Ves a lo que me refiero?

Imagínate esto: estás revisando una utilería que usa "los más prometedores algoritmos de inteligencia artificial" para encontrar y solucionar problemas de concurrencia. Lo inicias e inmediatamente notas que han escrito mal "inteligencia" en la pantalla de inicio. Un poco optimista, pensarás: es sólo un error de dedo, ¿verdad? Entonces notas que la pantalla de configuración usa varias casillas que deberían ser botones de radio y algunos de los atajos de teclado no funcionan. Ahora bien, ninguno de estos son un gran problema, pero conforme los errores se van sumando empiezas a preguntarte sobre los programadores. Si no pueden tener las cosas sencillas bien, ¿cuáles son las probabilidades de que su IA pueda realmente encontrar y solucionar algo tan complicado como los problemas de concurrencia?

Puede que sean genios quienes estaban tan enfocados a hacer la IA increíblemente mejor como para no notar esas pequeñas cosas triviales. Y sin esos "testers exigentes" apuntando los problemas, terminaste encontrándolos. Ahora te estás cuestionando la competencia de los programadores.

Así que por extraño que suene, estos ***testers***, quienes parecen determinados a exponer cada pequeño error en tu código, son realmente tus amigos.

Un binario

Por Steve Freeman · Traducción: Espartaco Palma

He visto muchos proyectos en los cuales la compilación reescribe alguna parte del código para generar un binario personalizado para cada ambiente destino. Esto siempre hace las cosas más complicadas de lo que deberían ser, e introduce el riesgo de que el equipo podría no tener versiones consistentes en cada instalación. Como mínimo involucra la compilación de múltiples, casi idénticas copias de software, cada una tiene que ser desplegada en el lugar correcto. Significa más partes movibles de lo necesario, lo que significa más oportunidad de cometer un error.

Una vez trabajé en un equipo en el cual cada cambio tenía que ser revisado en cada ciclo de compilación, por los que los testers se quedaban esperando cada que se necesitaba un ajuste menor (¿mencioné que la compilación tomaba también mucho tiempo?). También trabajé en un equipo en el que los administradores de sistemas insistían en reconstruir desde cero en producción (usando el mismo script que hicimos), lo que significaba que no teníamos pruebas de que la versión en producción era la misma que había estado bajo prueba. Y así por el estilo.

La regla es sencilla: compila un sólo binario que puedas identificar y promover a través de todas las etapas en la línea de liberación. Mantén detalles específicos del entorno en el ambiente. Esto podría significar, por ejemplo, mantenerlos en el contenedor de componentes, en un archivo conocido o en la ruta.

Si tu equipo tiene un revoltijo de código para compilar o almacenar todas las configuraciones destino en el código, esto sugiere que nadie ha pensado el diseño con el suficiente cuidado para separar estas características, que son fundamentales de la aplicación, de aquellas que son específicas de las plataforma. O podría ser peor: el equipo sabe qué hacer, pero no puede priorizar el esfuerzo para hacer el cambio.

Por supuesto, hay excepciones: podrías estar compilando para algún destino que tiene importantes restricciones de recursos, pero esto no aplica para la mayoría de nosotros que estamos escribiendo aplicaciones de “bases de datos a pantalla y de regreso”. Alternativamente, podrías estar viviendo con algún desorden heredado que es muy difícil de corregir ahora mismo. En tales casos, tienes que mover gradualmente, pero empezar tan pronto como sea posible.

Una cosa más: mantén la información del entorno con algún control de versiones. No hay nada peor que romper una configuración de entorno y no ser capaz de imaginarte qué cambió. La información de entorno debería ser versionada separadamente del código, ya que cambiará a diferentes periodos y por diferentes razones. Algunos equipos usan sistemas de control de versiones distribuidos para esto (como bazaar y git), ya que hacen más fácil de enviar cambios hechos en ambientes de producción –como sucede inevitablemente– de vuelta al repositorio.

CONSEJO 62 DE 97

Sólo el código dice la verdad

Por Peter Sommerlad · Traducción: Espartaco Palma

La semántica final de un programa está dada por el código que se ejecuta. ¡Si esto es únicamente en formato binario, será una lectura difícil! El código fuente debe, sin embargo, estar disponible si se trata de tu programa, cualquier desarrollo de software comercial típico, un proyecto de software libre o código en un lenguaje interpretado de forma dinámica. Al mirar el código fuente, el significado del programa debería ser evidente. Para saber qué hace el programa, el código es, en última instancia, de lo que puedes estar seguro. Hasta el documento de requisitos más preciso no dice toda la verdad: no contiene el relato detallado de lo que el programa está haciendo, sólo las intenciones de más alto nivel del analista de requerimientos. Un documento de diseño podría capturar un diseño planeado, pero carece del nivel necesario de detalle de la implementación. Estos documento pueden perder sincronía con la implementación actual... o simplemente se han perdido. O nunca fueron escritos, en primer lugar. El código fuente puede ser lo único que queda.

Con esto en mente, pregúntate: ¿qué tan claro es tu código al decirte a ti o a cualquier otro programador qué es lo que está haciendo?

Podrías decir: “Oh, mis comentarios te dirán todo lo que necesitas saber”. Pero recuerda que los comentarios no son código en ejecución. Pueden ser tan malos como cualquier otra forma de documentación. Existe una tradición que dice que los comentarios son incondicionalmente algo bueno, así que algunos programadores escriben más y más comentarios, incluso reiniciando y explicando trivialidades que son obvias en el código. Ésa es la forma errónea de clarificar tu código. Si tu código tiene comentarios, considera **refactorizar** para que no los tenga. Los comentarios extensos pueden saturar el espacio en la pantalla e incluso pueden ser ocultados automáticamente por tu `IDE`. Si necesitas explicar un cambio, hazlo en el mensaje de confirmación del sistema de control de versiones, no en el código.

¿Qué se puede hacer para hacer que tu código diga la verdad lo más claro posible? Lucha por buenos nombres. Estructura tu código con respecto a la funcionalidad cohesiva, que también facilita la nomenclatura. Desacopla el código para conseguir ortogonalidad. Escribe **pruebas automatizadas** explicando el comportamiento previsto y comprueba las interfaces. Refactoriza sin piedad cuando aprendas

cómo codificar una solución mejor y más sencilla. Haz que tu código sea tan sencillo como sea posible para leer y entender.

Trata a tu código como a cualquier otra composición, como un poema, un ensayo, un blog público o un email importante. Elabora lo que expresas con cuidado, de modo que haga lo que debe y comunique tan directamente como sea posible lo que está haciendo, para que comunique tus intenciones cuando no estés. Recuerda que el código útil se usa mucho más tiempo de lo previsto. Los programadores de mantenimiento te lo agradecerán. Y, si eres un programador de mantenimiento y el código en el que estás trabajando no dice la verdad fácilmente, aplica las directrices anteriores de manera proactiva. Establece algo de cordura en el código y mantén tu propia cordura.

Adueñate (y Refactoriza) la compilación

Por Steve Berczuk · Traducción: Espartaco Palma

No es poco común para los equipos que, aunque son altamente disciplinados sobre las prácticas de codificación, descuiden los scripts de compilación, quizás por la creencia de que son meramente un detalle de poca importancia o por el miedo de que son complejos y necesitan ser atendidos por el culto de la ingeniería de la liberación. Los scripts que no son posibles de mantener, con duplicaciones y errores, causan problemas de la misma magnitud que aquellos con código pobremente factorizado.

Una de las razones por las que los desarrolladores hábiles y disciplinados tratan la compilación como algo secundario es que los scripts de compilación son frecuentemente escritos en un lenguaje diferente al código fuente. Otra es que la compilación no es realmente "código". Estas justificaciones van en contra de la realidad de que la mayoría de los desarrolladores de software disfrutan aprendiendo nuevos lenguajes y que la compilación es lo que crea artefactos ejecutables para desarrolladores y usuarios finales para probar y ejecutar. El código es inútil si no ha sido compilado, y la compilación es lo que define el componente de arquitectura de la aplicación. La compilación es una parte esencial del desarrollo, y las decisiones sobre el proceso compilado pueden hacer más simples tanto el código como la codificación.

Los scripts para la compilación que son escritos usando modismos erróneos son difíciles de mantener y, más importante, de mejorar. Vale la pena tomarse tiempo para entender la forma correcta de realizar un cambio. Los errores pueden aparecen cuando una aplicación se compila con la versión incorrecta de una dependencia o cuando la configuración del tiempo de compilador está mal.

Tradicionalmente las pruebas han sido algo que siempre fue dejado al equipo de "Quality Assurance". Ahora nos damos cuenta de que hacer pruebas mientras codificamos es necesario para permitirnos liberar el valor predeciblemente. Del mismo modo, el proceso de compilación tiene que ser propiedad del equipo de desarrollo.

Entender la compilación puede simplificar el ciclo de vida completo y reducir costos. Una compilación simple de ejecutar permite al nuevo desarrollador empezar rápida y fácilmente. La automatización de la configuración de compilación puede permitirte obtener resultados consistentes cuando muchas personas están trabajando en un proyecto, evitando el "a mí me funciona". Muchas herramientas para compilación te permiten ejecutar reportes de calidad de código, lo que hace posible detectar problemas potenciales tempranamente. Al entender cómo hacer tuya la compilación, puedes ayudarte a ti mismo y a los integrantes de tu equipo. Enfócate en codificar características, en beneficio de las partes interesadas y para hacer tu trabajo más agradable.

Aprende lo suficiente de tu proceso de compilación para saber cuándo y cómo realizar los cambios. Los scripts de compilación son código. También son muy importantes para dejárselos a alguien más, la aplicación no está completa hasta que se compila. El trabajo de programación no está completo hasta que hayamos liberado software funcionando.

Programa en pareja y siente el flujo

Por Gudny Hauknes, Ann Katrin Gagnat, y Kari Røssland · Traducción: Espartaco Palma

Imagina que estás totalmente absorto en lo que estás haciendo, enfocado, dedicado e involucrado. Pudiste haber perdido el rastro del tiempo. Probablemente te sientas feliz. Estás experimentando el flujo. Es difícil alcanzar y mantener el flujo de todo el equipo de desarrolladores debido a que hay tantas interrupciones, interacciones y otras distracciones que puede ser roto fácilmente.

Si ya has practicado la programación en pareja, probablemente estás familiarizado con cómo el emparejamiento contribuye al flujo. Si no lo estás, ¡queremos usar nuestras experiencias para motivarte a comenzar ahora mismo! Para tener éxito con la programación en pares tanto los miembros individuales del equipo y el equipo como un todo tienen que poner algo de esfuerzo.

Como miembro de un equipo, sé paciente con los desarrolladores menos experimentados que tú. Enfrenta tus miedos de ser intimidado por desarrolladores más hábiles. Date cuenta de que la gente es diferente y valóralo. Sé consciente de tus propias fortalezas y debilidades, así como las de los otros miembros del equipo. Podrías sorprenderte de cuánto puedes aprender de tus colegas.

Como equipo, introduce la programación en pareja para promover la distribución de habilidades y conocimiento a través del proyecto. Deberás solucionar tus tareas en parejas y rotar las parejas y tareas frecuentemente. Acordar una regla de rotación. Pongan la regla a un lado o ajústenla cuando sea necesario. Nuestra experiencia es que no necesariamente tienen que completar una tarea antes de rotarla a otro par. Interrumpir las tareas para pasarla a otra pareja puede sonar contradictorio, pero hemos descubierto que funciona.

Existen numerosas situaciones en las que el flujo se puede romper, pero aquí es donde la programación en pareja ayuda a mantenerlo:

> **Reduce el “factor camión”**: es un experimento mental ligeramente mórbido, pero ¿cuántos de tus miembros del equipo tendrían que ser golpeados por un camión antes de que el equipo sea incapaz de completar la entrega final? En otras palabras, ¿qué tan dependiente es tu entrega de ciertos miembros del equipo? ¿El conocimiento se privilegia o se comparte? Si has estado rotando las tareas entre las parejas, siempre hay alguien más que tiene el conocimiento y puede completar el trabajo. El flujo del equipo no es afectado por el “factor camión”.
>
> **Soluciona problemas efectivamente**: si estás programando en pareja y entras en un problema difícil, siempre tendrás a alguien con quién discutirlo. Este diálogo puede abrir más posibilidades que si estás atorado tú solo. Conforme el trabajo se rota, tu solución será revisitada y reconsiderada por el siguiente par, así que no importa si inicialmente no elegiste la solución óptima.

Integra sin problemas: si tu tarea actual consiste en llamar otro fragmento de código, esperas que los nombres de los métodos, los documentos y las pruebas sean lo suficientemente descriptivas para darte una idea de lo que hacen. Si no, hacer pareja con un desarrollador que estaba involucrado en escribir ese código te dará una mejor vista general e integración rápida con tu propio código. Adicionalmente, puedes usar la discusión como una oportunidad para mejorar la nomenclatura, documentos y pruebas.

Mitiga interrupciones: si alguien viene a preguntar algo, o suena tu teléfono, o tienes que contestar un correo urgente, o tienes que atender una reunión, tu socio de programación en pareja puede quedarse codificando. Cuando regreses, tu socio estará aún en el flujo y rápidamente lo alcanzarás y te incorporarás a él.

Los nuevos miembros de equipo aceleran rápidamente: con la programación en parejas, y con una adecuada rotación de pares y tareas, los recién llegados conocerán rápidamente tanto el código como a los otros miembros del equipo.

El flujo te hace increíblemente productivo. Pero también es vulnerable. Haz lo que puedas para obtenerlo y aférrate a él cuando lo tengas.

CONSEJO 65 DE 97

Da preferencia a tipos de Dominio Específico que los tipos primitivos

Por Einar Landre · Traducción: Espartaco Palma

El 23 de septiembre de 1999 el ***Mars Climate Orbiter*** de U$327.6 millones se perdió mientras entraba a la órbita alrededor de Marte, debido a un error del software aquí en la Tierra. Error que más tarde fue llamado de "métrica mixta". El software de la estación en tierra estaba trabajando en libras, mientras que la nave esperaba newtons, llevando a la estación a subestimar el poder de los propulsores de la nave en un factor de 4.45.

Éste es uno de los muchos ejemplos de fallas de software que se pudo haber prevenido, si se hubiera aplicado un ***tipado*** más fuerte y de dominio específico. Es también un ejemplo del razonamiento detrás de muchas características del lenguaje Ada, uno de sus principales metas de diseño era implementar software de seguridad crítica embebida. Ada estaba fuertemente ***tipado*** con revisiones estáticas de ambos: tipos primitivos y tipos definidos por el usuario.

EJEMPLO DE CÓDIGO

```
type Velocity_In_Knots is new Float range 0.0 .. 500.00;

type Distance_In_Nautical_Miles is new Float range 0.0 .. 3000.00;

Velocity: Velocity_In_Knots;

Distance: Distance_In_Nautical_Miles;

Some_Number: Float;

Some_Number:= Distance + Velocity; -- Será capturado por el compilador como u
```

Los desarrolladores en dominios menos demandantes también se deberían beneficiar aplicando más ***tipado*** de dominio específico, en el que pudieran, de otro modo, continuar usando tipos de datos primitivos ofrecidos por el lenguaje y sus librerías, tales como cadenas y flotantes. En Java, C++, Python y otros lenguajes modernos, los tipos de datos abstractos son conocidos como clases. Usar clases como `Velocity_In_Knots` y `Distance_In_Nautical_Miles` agrega mucho valor con respecto a la calidad del código:

- El código se vuelve más legible conforme expresa conceptos de un dominio, no sólo flotantes o cadenas.
- El código se vuelve más ***testeable*** conforme encapsula su comportamiento, así es fácilmente probado.
- El código facilita la reutilización a través de aplicaciones y sistemas.

El enfoque es igualmente válido para usuarios de ambos lenguajes de tipo estático y dinámico. La única diferencia es que los desarrolladores que usan lenguajes de ***tipado*** estático obtienen más ayuda desde el compilados, mientras aquellos que adoptan lenguajes de ***tipado*** dinámico es más común que confíen en sus pruebas unitarias. El estilo de revisión podría ser diferente, pero la motivación y estilo de expresión no.

La moraleja es iniciar explorando los tipos de dominio específico con el fin de desarrollar software de calidad.

CONSEJO 66 DE 97

Evita errores

Por Giles Colborne · Traducción: Espartaco Palma

Los mensajes de error son la interacción más crítica entre el usuario y el resto del sistema. Suceden cuando la comunicación, entre el usuario y el sistema, está cerca del punto de quiebre.

Es fácil pensar que un error está siendo causado por una mala entrada de datos del usuario. Pero la gente comete errores de forma predecible y sistemática. Así que es posible depurar la comunicación entre el usuario y el resto del sistema así como lo harías con otros componentes del sistema.

Por ejemplo, digamos que quieres que el usuario introduzca una fecha en un rango permitido. En vez de dejar que el usuario introduzca cualquier fecha es mejor ofrecer un dispositivo, como una lista o calendario, mostrando sólo las fechas permitidas. Esto elimina cualquier oportunidad de que el usuario introduzca una fecha fuera del rango.

El formato del error es otro problema común. Por ejemplo, si a un usuario se le presenta un campo de texto como fecha e introduce una fecha ambigua como "Julio 29, 2012" es razonable el rechazarlo simplemente porque no es uno de los formatos preferidos (como "DD/MM/AAAA"). Es peor aún rechazar "29 / 07 / 2012" sólo porque contiene espacios extra; este tipo de problema es particularmente difícil de entender para usuarios porque la fecha parece estar en el formato deseado.

Este error ocurre porque es más fácil rechazar una fecha que analizar los tres o cuatro formatos de fecha más comunes. Este tipo de errores insignificantes llevan al usuarios a la frustración, que a su vez conduce a errores adicionales conforme el usuario pierde su concentración. En cambio, respeta las preferencias del usuario al entrar información, no los datos.

Otra forma de evitar errores de formato es ofrecer señales, por ejemplo, con una etiqueta dentro del campo mostrar el formato deseado ("DD/MM/AAAA"). Otra pista podría ser dividir el campo en tres cajas de texto de dos, dos y cuatro caracteres.

Las señales son diferentes de las instrucciones: las señales tienden a ser indicios; las instrucciones son detalladas; las señales ocurren en el punto de interacción; las instrucciones aparecen antes del punto de interacción. Las señales proveen contexto; las instrucciones dictan el uso.

En general, las instrucciones son ineficientes para prevenir errores. Los usuarios tienden a asumir que las interfaces trabajarán en la línea con su pasada experiencia ("¿seguramente todos saben el significado de 'Julio 29, 2012'?"). Así que las instrucción no son leídas. Las señales dan un suave codazo alejando a los usuarios del error.

Otra forma de evitar errores es ofrecer valores predeterminados. Por ejemplo, los usuarios típicamente introducen valores que corresponden al hoy, mañana, mi cumpleaños, mi fecha límite o la fecha que introduje la última vez que usé este formulario. Dependiendo del contexto, es probable que uno de ellos sea una buena opción de un valor predeterminado inteligente.

Sin importar la causa, los sistemas deberían ser tolerantes a errores. Puedes hacer esto proveyendo niveles múltiples de “deshacer” para todas las acciones y en especial las acciones que tenga el potencial de destruir o enmendar los datos del usuario.

El registro y análisis de las acciones de “deshacer” puede también ser un punto a destacar, en el cual la interfaz está atrayendo a los usuarios a errores inconscientes, tales como hacer clic persistentemente en un botón “equivocado”. Estos errores son, a menudo, causados por señales engañosas o secuencias de interacción que puedes rediseñar para prevenir más errores.

Cualquiera que sea el enfoque que tomes, la mayoría de los errores son sistemáticos, el resultado de malentendidos entre el usuario y el software. Entender cómo los usuarios piensan, interpretan información, toman decisiones e introducen datos, de entrada, te ayudará a depurar las interacciones entre el software y tus usuarios.

El Programador Profesional

Por Uncle Bob · Traducción: Espartaco Palma

¿Qué es un programador profesional?

El rasgo más importante de un programador profesional es la responsabilidad personal. Los programadores profesionales se responsabilizan por su carrera, sus estimaciones, el compromiso con su agenda, sus errores y su mano de obra. Un programador profesional no le pasa la responsabilidad a los demás.

> Si eres profesional, entonces eres responsable de tu propia carrera. Eres responsable de leer y aprender. Eres responsable de mantenerte actualizado con la industria y la tecnología. Muchos programadores piensan que es trabajo de sus patrones entrenarlos. Lo siento, están tremendamente equivocados. ¿Crees que los médicos se comportan de esa manera? ¿Crees que los abogados se comportan de esa manera? No, ellos se entrenan en su propio horario, y con su propio dinero. Ellos gastan muchas de sus horas libres leyendo revistas y tomando decisiones. Se mantienen al día. Y así debemos hacerlo nosotros. La relación entre tú y tu empleador está escrita claramente en tu contrato. En breve: prometen pagarte y tú prometes hacer un buen trabajo.
>
> Los profesionales asumen la responsabilidad del código que escriben. No liberan código a menos que sepan que funciona. Piensa en esto por un minuto. ¿Cómo puedes considerar llamarte profesional, si estás esperando liberar código del cuál no estás seguro? Los programadores profesionales esperan que QA no encuentre algo porque no liberan su código hasta que se ha probado completamente. Por supuesto, QA encontrará algunos problemas, debido a que nadie es perfecto. Pero, como profesionales, nuestra actitud debe ser: dejar nada para QA.
>
> Los profesionales son jugadores de equipo. Asumen responsabilidad de la salida de todo el equipo, no sólo de su propio trabajo. Se ayudan unos a otros, enseñan a los demás, aprenden unos de otros e, incluso, cubren a los demás, si es necesario. Cuando un compañero cae, los demás intervienen, sabiendo que algún día ellos van a ser los que necesiten cobertura.
>
> Los profesionales no toleran grandes listas de errores. Tener una lista así es ser descuidado. Los sistemas con cientos de ***issues*** en la base de datos de seguimiento de problemas son tragedias por la falta de cuidado. De hecho, en muchos proyectos, la propia necesidad de un sistema de seguimiento de problemas es un síntoma de descuido. Sólo los sistemas muy grandes deberían tener una lista de errores tan larga que sea necesario la automatización para manejarla.
>
> Los profesionales no hacen un desastre. Se enorgullecen de su mano de obra. Mantienen el **código limpio**, bien estructurado y fácil de leer. Siguen estándares acordados y las mejores prácticas. Ellos nunca, jamás se apresuran. Imagina que estás teniendo una experiencia “fuera de tu cuerpo” y miras a un cirujano realizar una cirugía a corazón abierto en ti. Este médico tiene un hora límite (en sentido literal). Debe terminar antes de que la máquina de derivación corazón-pulmón dañe muchas de las células sanguíneas. ¿Cómo quieres que se comporte? ¿Quieres que se comporte como el típico desarrollador de

> software, apresurado y haciendo un lío? ¿Quieres que diga: “regreso y lo arreglo luego”? ¿O quieres que se aferre cuidadosamente a sus disciplinas, tomándose su tiempo, seguro de que su enfoque es el mejor? ¿Quieres un desastre o profesionalidad?

Los profesionales son responsables. Asumen la responsabilidad por sus propias carreras. Asumen la responsabilidad de asegurarse de que su código funciona correctamente. Asumen la responsabilidad de la calidad de su mano de obra. No abandonan sus principios cuando los plazos se ciernen. De hecho, cuando la presión aumenta, los profesionales se aferran a las disciplinas que saben que son correctas.

Pon todo bajo Control de Versiones

Por Diomidis Spinellis · Traducción: Espartaco Palma

Pon todo lo que tienen tus proyectos bajo control de versiones. Los recursos que necesitas están ahí: herramientas libres como Subversion, Git, Mercurial y CVS; abundante espacio en disco; servidores baratos y poderosos; una red ubicua; e incluso servicios de hospedaje de proyectos. Después de instalar el software de control de versiones todo lo que necesitas para poner tu trabajo en su repositorio es ejecutar el comando apropiado en un directorio limpio que contenga tu código. Y sólo hay dos nuevas operaciones básicas por aprender: enviar el cambio en tus códigos al repositorio y actualizar tu directorio de trabajo a la versión del repositorio.

Una vez que el proyecto está bajo el control de versiones es obvio que puedes rastrear su historia, ver quién ha escrito qué código, y referir una versión del archivo o proyecto a través de un identificador único. Más importante, puedes hacer grandes cambios sin miedo; no más código comentado, sólo en caso de que lo necesites en el futuro, porque la versión anterior vive de manera segura en el repositorio. Puedes (y deberías) etiquetar una versión de software con un nombre simbólico, así podrás revisitarlo en el futuro en la versión exacta del software que tu cliente ejecuta. Puedes crear ramificaciones de desarrollo paralelo: la mayoría de los proyectos tienen una rama de desarrollo activo y una o varias más de mantenimiento de versiones publicadas que son apoyadas activamente.

Un sistema de control de versión minimiza la fricción entre desarrolladores. Cuando los programadores trabajan en partes diferentes del software esto se integra casi por arte de magia; cuando se empalma el código el sistema lo nota y permite que resuelvan los conflictos. Con un poco de configuración adicional el sistema puede notificar a todos los desarrolladores de cada cambio enviado, estableciendo un entendimiento común sobre el progreso del proyecto.

Al configurar el proyecto no seas tacaño: coloca todos los activos del proyecto bajo control de versiones. Además del código fuente, incluye la documentación, herramientas, scripts de creación, casos de prueba, obras de arte, e incluso bibliotecas. Con el proyecto completo y seguro en el repositorio (respaldado regularmente) se reduce al mínimo el daño de perder tu disco o datos. Configurar el ambiente de desarrollo en una máquina nueva consiste simplemente en traerse el proyecto desde el repositorio. Esto simplifica la distribución, construcción y las pruebas de código en diferentes plataformas: en cada máquina un simple comando de actualización se asegurará que el software está en la versión actual.

Una vez que ha visto la belleza de trabajar con un sistema de control de versiones, seguir unas cuantas reglas hará que tú y tu equipo sean más eficaces:

Enviar cada cambio lógico en una operación separada. Agrupar muchos cambios hará difícil desenredarlo en el futuro. Esto es especialmente importante al hacer una **refactorización** en todo el proyecto o cambios de estilo, los cuales pueden oscurecer otras modificaciones.

Acompañar cada envío con un mensaje explicativo. Como mínimo describir brevemente lo que ha cambiado, pero si también deseas grabar la justificación del cambio, entonces éste es el mejor lugar para almacenarlo.

Por último, no enviar código que rompa la construcción de un proyecto, de lo contrario se volverá impopular con los otros desarrolladores del proyecto.

La vida bajo un control de versión es demasiado buena como para arruinarla con errores fácilmente evitables.

CONSEJO 69 DE 97

Suelta el ratón y aléjate del teclado

Por Cay Horstmann · Traducción: Espartaco Palma

Te has enfocado por horas en algún raro problema y no hay solución a la vista. Así que te levantas para estirar las piernas o para llegar a la máquina expendedora y, en el camino de vuelta, la respuesta repentinamente se vuelve evidente.

¿Te suena familiar este escenario? ¿Alguna vez te preguntaste por qué sucede? El truco está en que mientras estás codificando, la parte lógica de tu cerebro está activa y el lado creativo se bloquea. No puede presentarte nada hasta que tu lado lógico tome un descanso.

Aquí está un ejemplo de la vida real: estaba limpiando un código heredado y me encontré con un método "interesante". Estaba diseñado para verificar que una cadena contenía una hora válida usando el formato ***hh:mm:ss xx***, donde ***hh*** representa la hora, ***mm*** representa los minutos, ***ss*** representa segundos y ***xx*** podría ser AM o PM.

El método utilizaba el siguiente código para convertir dos caracteres (representando la hora) en un número y verificando que estuviera en el rango adecuado: :

EJEMPLO DE CÓDIGO

```
try {
    Integer.parseInt(time.substring(0, 2));
} catch (Exception x) {
    return false;
}

if (Integer.parseInt(time.substring(0, 2)) > 12) {
    return false;
}
```

El mismo código aparecía dos veces más, con cambios apropiados para el carácter y el límite superior, para poner a prueba los minutos y segundos. El método terminaba con estas líneas para comprobar AM y PM.

EJEMPLO DE CÓDIGO

```
if (!time.substring(9, 11).equals("AM") &
    !time.substring(9, 11).equals("PM")) {
    return false;
}
```

Si ninguna de esta serie de comparaciones fallaba, regresando `false`, el método regresaba `true`.

Si el código anterior se ve confuso y difícil de seguir, no te preocupes. Yo también lo creía, lo que significaba que había encontrado algo digno de limpieza. Lo refactoricé y escribí unas cuantas pruebas unitarias, sólo para estar seguro de que aún funcionaba

Cuando terminé, me sentía satisfecho con el resultado. La nueva versión era fácil de leer, de la mitad del tamaño y más precisa debido a que el código original sólo probaba los límites superiores de las horas, minutos y segundos.

Mientras me preparaba para trabajar al día siguiente, una idea surgió en mi cabeza: ¿por qué no validar la cadena usando una expresión regular? Después de unos minutos escribiendo, tenía una implementación funcional de sólo una línea de código. Aquí está:

EJEMPLO DE CÓDIGO

```
public static boolean validateTime(String time) {
    return time.matches("(0[1-9]|1[0-2]):[0-5][0-9]:[0-5][0-9] ([AP]M)");
}
```

El punto de esta historia no es que eventualmente reemplacé cerca de 30 líneas de código con sólo una. El punto es que hasta que me alejé de la computadora pensaba que mi primer intento era la mejor solución al problema.

Así que la próxima vez que estés ante un problema desagradable, hazte un favor: una vez que realmente entiendas el problema ve a hacer algo que involucre el lado creativo de tu cerebro; esboza el problema, escucha algo de música o da un paseo al aire libre. A veces la mejor cosa que puedes hacer para resolver un problema es soltar el ratón y alejarte del teclado.

Lee el código

Por Karianne Berg · Traducción: Espartaco Palma

Nosotros, los programadores, somos criaturas raras. Amamos escribir código. Pero cuando toca leerlo usualmente nos asustamos. Después de todo, escribir código es más divertido y leerlo es difícil, algunas veces casi imposible. Leer el código de otras personas es particularmente difícil. No porque el código de las demás personas sea malo, sino porque piensan y solucionan problemas de una manera diferente a la tuya. ¿Alguna vez consideraste que leer el código de alguien más podría mejorar el tuyo?

La próxima vez que leas algún código, detente y piensa un momento. ¿El código es fácil o difícil de leer? Si es difícil de leer, ¿por qué lo es? ¿Su formato es pobre? ¿Está nombrado inconsistente o ilógicamente? ¿Mezcla muchas preocupaciones en la misma pieza de código? ¿Quizá la elección del lenguaje impide que el código sea legible? Trata de aprender de los errores de la gente, así tu código no contendrá los mismos. Puedes recibir unas cuantas sorpresas. Por ejemplo, las técnicas de ruptura de dependencias puede ser buenas para bajos acoplamientos, pero a veces puede hacer que el código sea difícil de leer. Lo que algunas personas llaman código elegante, otros lo llaman ilegible.

Si el código es fácil de leer, detente para ver si hay algo útil que puedas aprender. Quizás hay un patrón de diseño en uso que no conocías o que habías luchado para poder implementar. Tal vez los métodos son más cortos y sus nombres más expresivos que los tuyos. Algunos proyectos de código abierto están llenos de buenos ejemplos de cómo escribir código brillante y legible, ¡mientras otros sirven de ejemplo de todo lo contrario! Revisa un poco de su código y da un vistazo.

Leer código antiguo de algún proyecto que no estás trabajando actualmente también puede ser una experiencia enriquecedora. Inicia con algunos de tus más viejos programas y avanza hacia el código presente. Probablemente encontrarás que no es del todo fácil leerlo como cuando lo escribiste. Tu código de un principio podría también tener un cierto valor de entretenimiento embarazoso, como cuando se te recuerdan todas las cosas que dijiste mientras estabas bebiendo en la cantina la noche anterior. Mirar cómo has desarrollado tus habilidades a lo largo de los años puede ser realmente motivador. Observa qué áreas del código son difíciles de leer y considera si todavía estás escribiendo código del mismo modo hoy en día.

Así que la próxima vez que sientas la necesidad de mejorar tus habilidades de programación, no leas otro libro. Lee el código.

Lee las humanidades

Por Keith Braithwaite · Traducción: Espartaco Palma

En todo y hasta en el proyecto más pequeño de desarrollo, las personas trabajan con personas. En todos y hasta en el campo más abstracto de investigación, las personas escriben software para las personas a las que ayudan en alguna de sus metas. La gente escribe software con gente para la gente. Es un negocio de personas. Desafortunadamente, lo que se enseña a programadores, a menudo, los equipa muy mal para hacer frente a las personas con las que trabajan. Afortunadamente hay un completo campo de estudio que puede ayudar.

Por ejemplo, Ludwig Wittgenstein plantea un buen caso en “Philosophical Investigations” (y donde sea): cualquier lenguaje que usamos para hablarnos no es, no puede ser, una formato de serialización para llevar un pensamiento o idea o imagen de la cabeza de una persona a otra. Ya deberíamos estar en guardia en contra del malentendido cuando “obtenemos requerimientos”. Wittgenstein también muestra que nuestra habilidad para entendernos del todo no surge de definiciones compartidas, surge de una experiencia compartida, de una forma de vida. Esto puede ser una razón por la cuál los programadores que están inmersos en su dominio del problema tienden a hacerlo mejor que aquellos que están fuera de ello.

Lakoff y Johnson nos presentan un catálogo de “Metáforas por la que vivimos”, sugiriendo que el lenguaje es ampliamente metafórico, y que esas metáforas ofrecen una percepción de cómo podemos entender el mundo. Incluso los términos aparentemente concretos, como “flujo de efectivo”, que podríamos encontrar en una plática sobre el sistema de finanzas, pueden ser vistos como metafóricos: “el dinero es un fluido”. ¿Cómo se hace que la metáfora influya en la forma en que pensamos sobre los sistemas que manejan dinero? O podríamos hablar acerca de capas en una pila de protocolos, como algunos de alto nivel y otros de bajo nivel. Algo poderosamente metafórico: el usuario está “arriba” y la tecnología está “caída”. Esto expone nuestro pensamiento acerca de la estructura de los sistemas que construimos. Esto puede también marcar un flojo hábito de pensamiento del que nosotros deberíamos beneficiarnos de vez en cuando.

Martin Heidegger estudió de cerca la manera en que la gente experimenta herramientas. Los programadores construyen y usan herramientas, pensamos en ello y creamos, modificamos y recreamos la herramienta. Las herramientas son objeto de interés para nosotros. Pero para sus usuarios, como Heidegger muestra en “Being and Time”, una herramienta se convierte en una cosa invisible entendida sólo en su uso. Para los usuarios las herramientas sólo se convierten en objetos de interés cuando no funcionan. Esta diferencia en énfasis es útil para tomar en cuenta cuando la usabilidad está a discusión.

Eleanor Rosch anuló el modelo aristotélico de las categorías en las que organizamos nuestra comprensión del mundo. Cuando los programadores preguntan a los usuarios sobre su deseos para con un sistema, ellos tienden a preguntar las definiciones a través de predicados. Esto es muy conveniente para nosotros. Los términos en predicado pueden ser convertidos fácilmente en atributos de una clase o columnas en una table.

Este tipo de categorías son difíciles de entender, disjuntas y ordenas. Desafortunadamente, tal y como Rosh mostró en “Natural Categories” y trabajos posteriores, la gente no entiende el mundo, en general, de esta forma. Ellos lo entienden en formas basadas en ejemplos. Algunos ejemplos, como los tan llamados prototipos, son mejores que otros y, por lo tanto, las categorías resultantes son difusas, se superponen y pueden tener una rica estructura interna. Mientras sigamos insistiendo en respuestas Aristotélicas seremos incapaces de preguntar a los usuarios las preguntas correctas sobre su mundo, y estaremos luchando por llegar al común entendimiento que necesitamos.

CONSEJO 72 DE 97

Reinventa la rueda frecuentemente

Por Jason P Sage · Traducción: Espartaco Palma

IDEA CLAVE

"Sólo tienes que utilizar algo existente, es una tontería reinventar la rueda..."

¿Alguna vez has oído esto o alguna variación? ¡Seguro que sí! Todos los desarrolladores y estudiantes probablemente han escuchado un comentario como éste con frecuencia. ¿Por qué pensarlo? ¿Por qué reinventar la rueda es tan mal visto? Porque, con frecuencia o no, el código existente es código en funcionamiento. Ya ha pasado por algún tipo de control de calidad, pruebas rigurosas y se está utilizando con éxito. Adicionalmente, el tiempo y esfuerzo invertido en la reinvención es poco probable que pague tan bien como usar producto o **código base** existente. ¿Deberías preocuparte por reinventar la rueda? ¿Por qué? ¿Cuándo?

Quizá has visto publicaciones sobre patrones en el desarrollo de software o libros sobre diseño de software. Estos libros pueden ser aburridos independientemente de la maravillosa información contenida en ellos. Del mismo modo en que ver una película sobre navegación es muy diferente a salir a navegar, así también es usar código existente frente a diseñar tu propio software desde cero, probándolo, rompiéndolo, reparándolo y mejorándolo a lo largo del camino.

Reinventar la rueda no es sólo un ejercicio en dónde colocar constructos de código: se trata de cómo conseguir un conocimiento profundo del funcionamiento interno de varios componentes que ya existen. ¿Sabes cómo funcionan los gestores de memoria? ¿La paginación virtual? ¿Podrías implementarlo por ti mismo? ¿Qué tal las listas doblemente enlazadas? ¿Clases de matrices dinámicas? ¿Clientes ODBC? ¿Podrías escribir una interfaz gráfica de usuario que funcione como alguna otra muy popular que conozcas o te guste? ¿Puedes crear tu propios ***widgets*** de navegador web? ¿Sabes cuándo escribir un sistema multiplexado contra uno multihilo? ¿Cómo decidir entre una base de datos basada en archivos o en memoria? La mayoría de los desarrolladores nunca han creado estas implementaciones de software central por sí mismos y, por lo tanto, no tienen un conocimiento profundo de cómo funcionan. Como consecuencia, todo este tipo de software es visto como misteriosas cajas negras que simplemente funcionan. Comprender sólo la superficie del agua no es suficiente para revelarnos los peligros ocultos debajo de ella. No saber las cosas más profundas en el desarrollo de software limitará tu habilidad para crear trabajo estelar.

Reinventar la rueda y hacerlo mal es más valioso que equivocarse la primera vez. ¡Hay lecciones aprendidas en la prueba y error que tienen un componente emocional que la lectura de un libro técnico no te

puede ofrecer!

Los hechos aprendidos y la información en los libros son cruciales, pero convertirse en un gran programador es mucho más sobre adquirir experiencia que la recolección de hechos. Reinventar la rueda es tan importante para la educación y habilidades de un desarrollador como lo es el levantamiento de pesas para el fisicoculturista.

Resiste la tentación del patrón Singleton

Por Sam Saariste · Traducción: Espartaco Palma

El patrón Singleton resuelve muchos de tus problemas. Sabes que sólo necesitas una sola instancia. Tienes una garantía de que esta instancia fue inicializada antes de ser usada. Mantiene tu diseño simple y con un sólo punto de acceso global. Todo está bien. ¿Qué es lo que no me gusta de este clásico patrón de diseño?

Pues mucho, resulta ser. Puede ser tentador, pero la experiencia me mostró que la mayoría de los ***Singleton*** hacen realmente más daño que bien. Dificultan las pruebas y dañan la capacidad de mantenimiento. Desafortunadamente, esta sabiduría adicional no está tan propagada como debería y los Singletons continúan siendo irresistibles para la mayoría de los programadores. Pero vale la pena resistirse:

> El requisito de instancia única es con frecuencia imaginado. En muchos casos es pura especulación de que no se necesitarán instancias adicionales en el futuro. Difundir tales propiedades especulativas, a través del diseño de la aplicación, está destinado a causar dolor en algún momento. Los requerimientos cambiarán. El buen diseño lo adopta. Los ***Singleton*** no.
>
> Los ***Singleton*** causan dependencia implícita entre unidades de código conceptualmente independientes. Esto es problemático porque están ocultos e introducen acoplamiento innecesario entre las unidades. Este olor del código se pudre cuando intentas escribir pruebas unitarias, las cuales dependen de soltar el acoplamiento y de la habilidad de sustituir selectivamente una implementación simulada (***mock***) de una real. Los ***Singleton*** previenen la simulación directa.
>
> Los ***Singleton*** también llevan estado persistente implícito, lo que dificulta más las pruebas unitarias. Las pruebas unitarias dependen de que sean independientes entre sí, así las pruebas pueden ser ejecutadas en cualquier orden y el programa puede ser configurado a un estado conocido antes de la ejecución de cada prueba unitaria. Una vez que hayas introducido ***Singleton*** con estado mutable, esto puede ser más difícil de llevar a cabo. Además, dicho estado globalmente accesible hace más difícil razonar sobre el código, especialmente en ambientes multihilos.
>
> Los multihilos introducen futuras fallas en el patrón ***Singleton***. El bloqueo directo al acceso no es muy eficiente, así es como el llamado patrón de doble revisión de bloqueo (DCLP) ha ganado popularidad. Desafortunadamente, esto puede llevar una forma adicional de atracción fatal. Resulta ser que muchos lenguajes DCLP no son ***thread-safe*** e, incluso cuando lo son, aún hay oportunidades de sutiles errores.

El limpiado de ***Singleton*** puede presentar un reto final:

> No hay soporte para matar explícitamente a un ***Singleton***, lo cuál puede ser un problema delicado en algunos contextos. Por ejemplo, en una arquitectura de ***plug-ins*** en la que un ***plug-in*** sólo puede ser descargado de forma segura después de que todos sus objetos han sido limpiados.

No hay orden implícito de limpieza de ***Singleton*** al salir del programa. Esto puede ser problemático para aplicaciones que contienen ***Singleton*** con interdependencias. Cuando se cierra dicha aplicación, un ***Singleton*** puede acceder a otra que ya ha sido destruida.

Algunas de estas deficiencias pueden ser superadas mediante la introducción de mecanismos adicionales. Sin embargo, esto viene con el costo de complejidad adicional en código que se podría haber evitado escogiendo un diseño alternativo.

Por lo tanto, restringe el uso del patrón ***Singleton*** a las clases que realmente nunca deber ser instanciadas más de una vez. No uses un ***Singleton*** como punto de acceso global desde código arbitrario. En vez de ello, el acceso directo al ***Singleton*** debería ser desde sólo unos pocos lugares definidos, donde pueda dársele vuelta vía una interfaz hacia otro código. Este otro código no lo sabe y así no depende de si un ***Singleton*** o cualquier otro tipo de clase implementa la interfaz. Esto rompe la dependencia que impide las pruebas unitarias y mejora la capacidad de mantenimiento. Así que, la próxima vez que estés pensando en implementar o acceder a un ***Singleton*** espero que hagas una pausa y lo pienses de nuevo.

El camino al mejor rendimiento está lleno de sucias bombas de código

Por Kirk Pepperdine · Traducción: Espartaco Palma

Más frecuentemente que nunca, la optimización de rendimiento en un sistema requiere que alteres código. Cuando tenemos que alterar código, cada porción intrincadamente compleja o altamente acoplada es una sucia bomba de código, en espera de descarrilar el esfuerzo. La primera víctima de código sucio será tu agenda. Si el camino a seguir es suave, será fácil predecir cuando acabará. Los encuentros inesperados con el código sucio harán que sea muy difícil hacer una predicción cuerda.

Considera la situación en la que encuentras un punto de ejecución complicado. El curso normal de acción es reducir la fortaleza del algoritmo en cuestión. Digamos que respondes con "3-4 horas" a un estimado que te pide el gerente. Si aplicas el ***fix*** te darás cuenta rápidamente que has descompuesto una parte dependiente. Debido a que las cosas están relacionadas, a menudo están necesariamente acopladas, estas descomposturas son esperadas y se cuenta con ellas. Pero, ¿qué pasa si un arreglo en esa dependencia termina rompiéndose en otra parte dependiente? Por otro lado, entre más lejos está la dependencia de su origen, menos probable es reconocerla como tal y tomarla en cuenta en tu estimado. De repente tu estimado de 3-4 horas pueden elevarse fácilmente a 3-4 semanas. Con frecuencia esta inflación inesperada en la agenda sucede 1 o 2 días, todas al mismo tiempo. No es raro el ver refactorizaciones "rápidas" que eventualmente toman varios meses en ser completadas. En esos casos, el daño en la credibilidad y capital político del equipo responsable variará de severo a terminal. Si tan sólo tuviéramos una herramienta para ayudarnos a identificar y medir estos riesgos.

De hecho, tenemos varias maneras de medir y controlar el grado y profundidad de acoplamiento y complejidad de nuestro código. Las métricas de software puede ser usadas para contar las apariciones de característica específicas en nuestro código. Los valores de estos conteos se correlacionan con la calidad del código. Dos de estas métricas que miden el acoplamiento son las llamadas ***fan-in*** y ***fan- out***. El ***fan-out*** está definido como el número de clases referenciadas, ya sea directa o indirectamente, para una clase en particular. Puedes pensar en esto como un recuento de todas las clases que deben ser compiladas antes de que tu clase pueda ser compilada. El ***fan-in*** un conteo de todas las clases que depende de una clase en específico. Conociendo el ***fan-out*** y ***fan-in*** podemos calcular un factor de inestabilidad usando I = fo / (fi + fo). Conforme se aproxima a 0, el paquete se vuelve más estable. En cuanto se aproxime a 1, el paquete se convierte en inestable. Los paquetes que son estables son objetivos de bajo riesgo, mientras que los

paquetes inestables son más propensos a estar llenos de sucias bombas de códigos. La meta de la **refactorización** es mover ***I*** lo más cercano a 0.

Cuando usamos métricas debemos recordar que sólo son reglas empíricas. Basándose puramente en las matemáticas puedes ver que el incremento de ***fi*** sin cambiar ***fo*** moverá ***I*** mas cerca a 0. Sin embargo hay una desventaja en tener el valor ***fan-in*** alto, pues estas clases serán más difíciles de modificar sin romper dependencias. Al no tener en cuenta el ***fan-out*** no estás reduciendo realmente el riesgo, por lo que debe aplicarse algún balance.

Una desventaja de las métricas de software es que la gran cantidad que números que producen las herramientas pueden ser intimidantes para los no iniciados. Dicho esto, las métricas de software pueden ser una poderosa herramienta en nuestra lucha por un **código limpio**. Pueden ayudar a identificar y eliminar las sucias bombas de código antes de que sean un serio riesgo al ejercicio de optimización del rendimiento.

La Simplicidad viene de la Reducción

Por Paul W. Homer · Traducción: Espartaco Palma

“Hazlo de nuevo…”, me dijo el jefe mientras su dedo presionaba con fuerza la tecla de borrado. Miré la pantalla de la computadora con una sensación de vacío muy familiar, mientras mi código –línea tras línea– desparecía en el olvido.

Mi jefe, Stefan, no siempre fue el más vocal de las personas, pero él sabía que era un mal código cuando lo veía. Y sabía exactamente qué hacer con él.

Había llegado a mi puesto actual como un programador estudiante con mucha energía, mucho entusiasmo y sin la menor idea de cómo codificar. Tenía esa horrible tendencia a pensar que la solución a cada problema era agregar otra variable en algún lugar. O escribir otra línea. En un mal día, en vez de que la lógica fuera haciéndose mejor con cada revisión, mi código se hacía gradualmente más grande, más complejo y mucho más lejos del trabajo consistente.

Es natural, sobre todo cuando estás apresurado, que sólo quieres hacer los menores cambios a un bloque de código existente, aunque sea horrible. Muchos programadores preservan mal código, temen que iniciar de nuevo requerirá mucho más esfuerzo que continuar donde se quedaron. Esto puede ser cierto para el código que está cerca de ser funcional, pero hay algunos códigos que están más allá de toda ayuda.

Se desperdicia más tiempo en tratar de salvar un mal código del que se debería. Una vez que algo se vuelve un sumidero de recursos, necesita ser descartado. Rápidamente

No es que debas tirar todo lo que has escrito, nombrado y formateado tan fácilmente. La reacción de mi jefe fue extrema, pero me obligó a repensar el código en el segundo (u ocasionalmente tercer) intento. Aún así, la mejor estrategia para arreglar un mal código es cambiándolo de tal modo que el código sea refactorizado sin misericordia, cambiado de lugar o borrado.

El código debería ser simple. Debería ser un mínimo de variables, funciones, declaraciones y otras necesidades sintácticas del lenguaje. Las líneas, variables adicionales… nada de adicional, en realidad, eso debería ser purgado. Removido inmediatamente. Lo que está ahí, lo que queda, sólo debería ser lo suficiente para realizar el trabajo, completar el algoritmo o realizar los cálculos. Cualquier otra cosa y todo lo demás es sólo ruido adicional no deseado, introducido accidentalmente y que obscurece el flujo. Ocultando las cosas importantes.

Por supuesto, si no lo logra, entonces sólo borra todo y escríbelo una vez más. Iniciar el diseño desde lo recordado a menudo puede ayudar a cortar una gran cantidad de desorden innecesario.

El Principio de Responsabilidad Única

Por Uncle Bob · Traducción: Espartaco Palma

Uno de los principios fundamentales de un buen diseño es: reúna las cosas que cambian por la misma razón y separe aquellas cosas que cambian por diferentes razones.

Este principio es conocido también como el Principio de la **Responsabilidad Única** o SRP (por sus siglas en inglés). En definitiva, se dice que un subsistema, módulo, clase o incluso una función no debe tener más de una razón para cambiar. El ejemplo clásico es una clase que tiene métodos relacionados con reglas de negocio, reportes y base de datos:

EJEMPLO DE CÓDIGO

```
public class Empleado {
  public Money calculaPago() ...
  public String reportaHoras() ...
  public void guardar() ...
}
```

Algunos programadores podrían pensar que poner estas tres funciones en la misma clase es perfectamente apropiado. Después de todo, se supone que las clases son colecciones de funciones que operan sobre las variables comunes. Sin embargo, el problema es que las tres funciones cambian por razones totalmente distintas. La función `calculaPago` cambiará cada vez que las reglas de negocio para calcular el pago cambien. La función `reportaHoras` cambiará cada vez que alguien quiera otro formato para el informe. La función `guardar` cambiará cada vez que los DBA cambien el esquema de base de datos. Estas tres razones de cambio se combinan para hacer a `Empleado` muy volátil. Cambiará por alguna de estas razones. Más importante aún, las clase que depende de `Empleado` será afectadas por estos cambios.

El buen diseño de sistemas significa que separamos el sistema en componentes que pueden ser implementados de forma independientemente. La implementación independiente significa que si cambiamos un componente no tenemos que volver a implementar alguno de los otros. Sin embargo, si `Empleado` es muy utilizado por muchas otras clases en otros componentes, entonces es probable que cada cambio a `Empleado` cause que los otros componentes tengan que volverse a implementar; negando así el mayor beneficio del diseño de componentes (o SOA, si se prefiere un nombre más de moda).

EJEMPLO DE CÓDIGO

```
public class Empleado {
  public Money calculaPago() ...
}

public class ReporteadorEmpleado {
  public String reportHora(Empleado e) ...
}

public class RepositorioEmpleado {
  public void guardar(Empleado e) ...
}
```

La simple división mostrada arriba resuelve estos problemas. Cada una de estas clases se puede colocar en un componente para sí mismas. O, mejor dicho, todos las clases de reporteo pueden ir en el componente de reporteo. Todas las clases relacionadas con base de datos pueden estar en el componente de repositorios. Y todas las reglas de negocios pueden entrar en el componente de reglas de negocio.

El lector astuto verá que aún existen dependencias en la solución anterior. Ese `Empleado` aún depende de las otras clases. Si se modifica `Empleado`, es probable que las otras clases se tengan que volver a compilar e implementar. Por lo tanto `Empleado` no se puede modificar y después implementar independientemente. Sin embargo, las otras clases pueden ser modificadas e implementadas independientemente. Ninguna modificación de alguna clase puede forzar a cualquiera de los otras a ser recompiladas o reimplementadas. Incluso `Empleado` podría ser implementada independientemente a través de un uso cuidadoso del Principio de Inversión de Dependencias (DIP), pero eso es un tema para otro libro.

La aplicación cuidadosa del SRP, separando las cosas que cambian por diferentes razones, es una de las claves para la creación de diseños que tienen una estructura de componentes de implementación independientemente.

Inicia con un Sí

Por Alex Miller · Traducción: Espartaco Palma

Recientemente fui a la tienda buscando arriba y abajo "edaname" (el cuál sólo sabía vagamente que era algún tipo de vegetal). No estaba seguro si era algo que podría encontrar en la sección de vegetales, la sección de congelados o en enlatados. Me rendí y busqué a una empleada para que me ayudara. ¡Ella tampoco sabía!

La empleada pudo haber respondido de muchas maneras distintas. Pudo haberme hecho sentir ignorante por no saber dónde buscar, darme vagas posibilidades o simplemente decirme que no lo tenían. Pero en vez de ello, usó el pedido como una oportunidad de encontrar una solución y ayudar al cliente. Llamó a otro empleado y en minutos me habían guiado al artículo deseado, ubicado en la sección de congelados.

La empleada en este caso miró en un pedido e inició con la premisa de que debería resolver el problema y satisfacer la petición. Inició con un sí, en vez de empezar con un no.

La primera vez que fui colocado en un rol de líder técnico, sentí que mi trabajo era proteger mi precioso software del ridículo flujo de demandas de los gestores de producto y analistas de negocio. Iniciaba muchas conversaciones viendo un pedido como algo que tenía que vencer, no algo que debía conceder.

En cierto punto, tuve una epifanía: quizás había una manera distinta de trabajar al cambiar mi perspectiva de iniciar con un no, iniciando con un sí. De hecho, he empezado a creer que iniciar con un sí es parte esencial de ser un líder tecnológico.

Este simple cambio radical alteró el cómo abordé mi trabajo. Como resultado, hay un montón de maneras de decir sí. Cuando alguien te dice: "Hey, esta aplicación sería mejor si hacemos todas las ventanas redondeadas y traslúcidas", puedes rechazarlo por ridículo. Pero frecuentemente es mejor iniciar con un "¿por qué?". En primer lugar, usualmente existe una actual e irresistible razón de por qué esa persona está pidiendo ventanas redondeadas y traslúcidas. Por ejemplo, quizás ustedes están a punto de firmar con un nuevo cliente muy grande con un comité de estándares que obliga a tener ventanas redondeadas y traslúcidas.

Constantemente encontrarás que cuando sabes el contexto de la petición, se abren nuevas posibilidades. Es común para la petición estar cumpliendo con el producto existente en alguna otra forma que permita decir sí sin trabajar: "De hecho, en las preferencias de usuario puedes descargar las cubiertas con ventanas traslúcidas y activarlas".

Algunas veces la otra persona simplemente no tendrá idea que lo encuentras incompatible con tu visión del producto. Me parece que es generalmente útil revertir ese "¿por qué?" hacia ti. Algunas veces el acto de expresar la razón hará más claro que tu primera reacción no tiene sentido. De lo contrario, quizá necesites elevarlo a un nivel superior de tomadores de decisiones. Recuerda, la meta de todo esto es decir sí a la otra persona e intentar hacerlo funcionar, no sólo por él, sino también por ti y tu equipo.

Si puedes expresar una irresistible explicación de por qué esa característica es incompatible con el producto existente, entonces es probable tener una conversación productiva sobre si están construyendo el producto

correcto. Sin importar cómo concluya esa conversación, todos se enfocarán más en qué es el producto y qué no lo es.

Iniciar con un sí significa trabajar con tus colegas, no contra ellos.

Retrocede y Automatiza, Automatiza, Automatiza

Por Cay Horstmann · Traducción: Espartaco Palma

Trabajé con programadores que, cuando se les pidió un conteo del número de líneas de código de un módulo, copiaban los archivos en un procesador de texto y usaban la característica de "número de líneas". Y lo hicieron de nuevo la siguiente semana y la semana siguiente. Fue malo.

Trabajé en un proyecto que tenía un proceso de implementación engorroso, implicaba la firma de código y mover el resultado a un servidor, requiriendo muchos clics con el ratón. Alguien lo automatizó y el script se ejecutó cientos de veces durante la prueba final, mucho más frecuentemente de lo previsto. Fue bueno.

Entonces, ¿por qué la gente realiza la misma tarea una y otra vez, en vez de retroceder y tomarse el tiempo de automatizarla?

Concepto erróneo común #1: La automatización es sólo para las pruebas

Seguro, la automatización en las pruebas es genial, pero ¿por qué detenerse ahí? Las tareas repetitivas están en cualquier proyecto: control de versiones, compilación, construcción de archivos JAR, generación de documentación, implementación y presentación de informes. Para muchas de estas tareas, el script es más poderoso que el ratón. Ejecutar tareas tediosas se convierte en algo más rápido y más fiable.

Concepto erróneo común #2: Tengo un IDE, así que no necesito automatizar

¿Alguna vez has tenido una discusión con un "pero (lo revisé | compila | pasa las pruebas) en mi máquina" con alguno de tus compañeros de equipo? Los `IDE` modernos tienen miles de configuraciones posibles y es prácticamente imposible asegurar que todos los miembros del equipo tienen configuraciones idénticas. Los sistemas de compilación automática, tales como Ant o Autotools, te proporcionan control y repetitividad.

Concepto erróneo común #3: Necesito aprender exóticas herramientas con el fin de automatizar

Puedes seguir con un lenguaje de shell decente (tales como bash o Powershell) y un sistema de automatización de compilación. Si necesitas interactuar con un sitio web, usa herramientas como iMacros o Selenium.

Concepto erróneo común #4: No puedo automatizar esta tarea porque no puedo manejar este tipo de formato

Si una parte de tu proceso requiere documentos Word, hojas de cálculo o imágenes, es cierto que puede ser un reto para la automatización, pero ¿es realmente necesario? ¿Puedes usar texto plano? ¿Valores separados por coma? ¿`XML`? ¿Alguna herramienta que genere un dibujo a partir de un archivo de texto? Con frecuencia, unos ligeros arreglos en el proceso puede llevar a un buen resultado con una dramática reducción del tedio.

Concepto erróneo común #5: No tengo el tiempo para averiguarlo

No tienes que aprender todo sobre bash o Ant para empezar. Aprende sobre la marcha. Cuando tengas una tarea que crees que pueda y deba ser automatizada, aprende sólo lo necesario acerca de la herramienta para hacerlo. Hazlo al inicio del proyecto cuando el tiempo es más fácil de encontrar. Una vez que has tenido éxito, tú y tu jefe verán que tiene sentido invertir en automatización.

Toma ventaja de las herramientas de análisis de código

Por Sarah Mount · Traducción: Espartaco Palma

El valor de las pruebas es algo que está siendo inculcado a los desarrolladores de software desde las primeras etapas de su jornada de programación. En años recientes el aumento de las pruebas unitarias, **desarrollo basado en pruebas** (test-driven), y los métodos ágiles han visto un surgimiento de interés en hacer más pruebas en todas las fases del ciclo de desarrollo. Sin embargo, las pruebas es sólo una de las muchas herramientas que puedes usar para mejorar la calidad del código.

Lejos, en la neblina del tiempo, cuando C era todavía un nuevo fenómeno, el tiempo de CPU y el almacenamiento de cualquier tipo eran un bien escaso. Los primeros compiladores de C eran conscientes de esto, así que reducían el número de pases que hacían a través del código quitando algunos análisis semánticos. Esto significaba que el compilador comprobaba sólo un pequeño subconjunto de errores que podían ser detectados al compilar. Para compensarlo, Stephen Johnson escribió una herramienta llamada ***lint*** –la cual remueve la pelusa de tu código– que implementaba algunos de los análisis estáticos que habían sido quitados por el compilador de C. Las herramientas de análisis estático, sin embargo, ganaron la reputación de obtener gran número de advertencias con falsos positivos y avisos sobre convenciones estilísticas que no siempre es necesario seguir.

El panorama actual de los lenguajes, compiladores y herramientas de análisis estático es muy diferente. La memoria y el tiempo de CPU ahora son relativamente baratos, por lo que los compiladores se puede permitir detectar más errores. Casi cualquier lenguaje cuenta con, al menos, una herramienta que comprueba violaciones de estilo, errores comunes y algunos errores astutos que pueden ser difíciles de capturar, tales como potenciales desreferencias de punteros nulos. Las herramientas más sofisticadas, como ***Split*** para C o ***Pylint*** para Python, son configurables, lo que significa que puedes escoger cuáles errores y advertencias emite la herramienta con un archivo de configuración, a través de la línea de comandos o en tu `IDE`. ***Splint*** incluso te permitirá anotar el código con comentarios que te dará mejores consejos sobre cómo funciona tu programa.

Si todo lo demás falla y te encuentras buscando errores simples o violación de normas que no son capturados por tu compilador, IDE o herramienta ***lint***, entonces siempre puedes llevar tu propio revisor estático. Esto no es tan difícil como suena. La mayoría de los lenguajes, particularmente aquellos etiquetados como dinámicos, exponen su árbol sintaxis abstracto y herramientas de compilación como parte de su biblioteca estándar. Vale la pena saber los rincones polvorientos de la biblioteca estándar que son usados por el equipo de desarrollo del lenguaje que estás usando, ya que frecuentemente contienen gemas ocultas que son útiles para análisis estático y pruebas dinámicas. Por ejemplo, la biblioteca estándar de Python contiene un

desensamblador que te dice el código bytecode usado para generar algún código compilado o código objeto. Esto suena como una herramienta obscura para escritores de compiladores en el equipo python-dev, pero es realmente útil para las situaciones diarias. Una cosa que puede desensamblar esta biblioteca es el último trazo de pila (***stack trace***), dándote una retroalimentación sobre exactamente cuál instrucción de ***bytecode*** lanzó la última excepción no capturada.

Así que no dejes que las pruebas sean el final de tu aseguramiento de calidad; toma ventaja de las herramientas de análisis y no tengas miedo de complicarte.

Prueba el comportamiento requerido, no el comportamiento incidental

Por Kevlin Henney · Traducción: Espartaco Palma

Una trampa común en las pruebas es asumir que lo que hace una implementación es precisamente lo que quieres probar. De primera instancia suena más como una virtud que una trampa. Sin embargo, expresado de otra manera, el tema se vuelve más obvio: una trampa común es escribir las pruebas para las especificaciones de una implementación, en las cuales estas especificaciones son incidentales y no tienen nada que ver con la funcionalidad deseada.

Cuando las pruebas están amarradas a las implementaciones incidentales, los cambios en la implementación son compatibles con el comportamiento requerido y pueden provocar que las pruebas fallen, conduciéndonos a falsos positivos. Los programadores frecuentemente responden ya sea reescribiendo los test o reescribiendo el código. Asumir que un falso positivo es realmente un verdadero positivo es frecuentemente una consecuencia de miedo, incertidumbre o duda. Al reescribir una prueba, los programadores o reenfocan la prueba al comportamiento requerido (bien) o simplemente lo amarran a la nueva implementación (mal). Las pruebas necesitan ser lo suficientemente precisas, pero también necesitan ser exactas.

Por ejemplo, en una comparación de tres vías, tales como el `strcpm` de C o el `String.compareTo` de Java, el requerimiento en el resultado es que es negativo si el lado izquierdo es menor que el derecho, positivo si el lado izquierdo es mayor que el derecho y cero si son considerados iguales. Este estilo de comparación es usado en muchas `API`, incluido el comparador de la función `qsort` de C y el `compareTo` en la interfaz `Comparable` de Java. Aunque los valores específicos -1 y +1 son comúnmente usados en la implementación para significar menor-que y mayor-que, respectivamente, los programadores a menudo asumen erróneamente que estos valores representan el requerimiento actual y consecuentemente escriben pruebas que clavan esta suposición en público.

Un tema similar surge cuando las pruebas que hacen asserts en el espaciado, texto exacto y otros aspectos del formato de texto y representación son incidentales. A menos que estés escribiendo, por ejemplo, un generador `XML` que ofrece un formateo configurable, el espaciado no debería ser significativo en la salida. Del mismo modo, amarrar el posicionamiento de botones y etiquetas en controles de Interfaz de Usuario (UI) reduce la opción de cambio y refina estas incidencias en el futuro. Los cambios menores en la

implementación, así como los cambios insignificantes en el formato se convierten de repente en cosas que rompen la compilación.

Las pruebas sobre-especificadas son frecuentemente un problema con enfoques de “caja blanca” en las pruebas unitarias. Las pruebas de “cajas blanca” usan la estructura del código para determinar los casos de prueba necesarios. La típica falla en las pruebas de “caja blanca” es que las pruebas terminan afirmando que el código hace lo que tiene que hacer. El sólo reiterar lo obvio no agrega valor y conduce a una falsa sensación de progreso y seguridad.

Para ser eficaces, las pruebas necesitan establecer obligaciones contractuales en lugar de parlotear la implementación. Necesitan tomar una visión de “caja negra” en las pruebas unitarias a probar, esbozando los contratos de la interfaz de manera ejecutable. Y, así, alinear el comportamiento probado con el comportamiento requerido.

Prueba precisa y concretamente

Por Kevlin Henney · Traducción: Espartaco Palma

Es importante probar el comportamiento deseado y esencial de una unidad de código, en vez de probar el comportamiento incidental de su implementación en particular. Pero esto no debería ser tomado, o mal tomado, como una excusa para las pruebas vagas. Las pruebas necesitan ser exactas y precisas.

Como ejemplo ilustrativo podemos tomar el caso intentado y probado múltiples veces, siendo ya todo un clásico: las rutinas de ordenamiento. Implementar un algoritmo de ordenamiento no es necesariamente una tarea diaria de un programador, pero el ordenamiento es una idea tan familiar que mucha gente cree saber qué esperar de ello. Esta familiaridad casual, sin embargo, puede hacer difícil ver más allá de ciertos supuestos.

Cuando se le pregunta a los programadores “¿qué es lo que probarías?”, la respuesta más común y, por mucho, es “el resultado del ordenamiento es una secuencia ordenada de elementos”. A pesar de que es verdad, no es toda la verdad. Cuando se les pide una condición más precisa, muchos programadores agregan que la frecuencia resultante debe ser de la misma longitud que el original. A pesar de que es correcto, aún no es suficiente. Por ejemplo, dada la siguiente secuencia:

> **IDEA CLAVE**
> 3 1 4 1 5 9

La siguiente secuencia satisface una poscondición de estar ordenado de manera no-descendiente y teniendo la misma longitud que la secuencia original:

> **IDEA CLAVE**
> 3 3 3 3 3 3

A pesar de que satisface las especificaciones, ¡esto es también algo a lo que ciertamente no nos referíamos! Este ejemplo está basado en un error tomado de un código de producción real (afortunadamente capturado antes de que fuera liberado), en el cual un simple desliz de un teclazo o un lapso momentáneo del razonamiento nos lleva a un elaborado mecanismo de llenar el resultado entero con el primer elemento de alguna matriz.

La poscondición completa es que el resultado esté ordenado y que tenga una permutación de los valores originales. Esto restringe apropiadamente el comportamiento requerido. Que la longitud del resultado sea el mismo que el de la longitud de la entrada viene con ello y no necesita ser reiniciado.

Aún estipular la poscondición en la manera descrita no es suficiente para darte una buena prueba. Una buena prueba debe ser leíble. Debe ser comprensible y suficiente como para que leyéndola puedas ver si es correcta (o no). A menos que ya tengas código por ahí para checar que una secuencia se ordena y que esa secuencia contiene una permutación de valores en otra, es muy probable que el código de prueba sea más complejo que el código a probar. Como Tony Hoare observa: "Hay dos manera de construir un diseño de software: una manera es hacerlo tan simple que obviamente no hay deficiencias y la otra es construirlo tan complicado que no hay deficiencias obvias".

Usando ejemplos concretos eliminamos esta complejidad accidental y oportunamente por accidente. Por ejemplo, dada la siguiente secuencia:

> **IDEA CLAVE**
>
> 3 1 4 1 5 9

El resultado del ordenamiento es el siguiente:

> **IDEA CLAVE**
>
> 1 1 3 4 5 9

Ninguna otra respuesta lo será. No aceptes sustitutos.

Los ejemplos concretos ayudan a ilustrar el comportamiento general de una manera accesible y no ambigua. El resultado de agregar un item a una colección vacía no es simplemente que no esté vacía: es que la colección ahora tiene un elemento y que ese elemento es el item agregado. Dos o más elementos calificarían como no vacío. Y estaría mal. Un sólo elemento de un valor diferente también estaría mal. El resultado de agregar una fila a una tabla no es simplemente que la tabla es una fila más grande. Esto también implica que la llave para la fila puede ser usada para recuperar la fila agregada. Y así por el estilo.

Al especificar el comportamiento, las pruebas deberían ser simplemente exactas: también deben ser precisas.

Haz pruebas mientras duermes (y los fines de semana)

Por Rajith Attapattu · Traducción: Espartaco Palma

Tranquilo. No me refiero a centros de desarrollo a larga distancia, horas extra en fin de semana o trabajar de noche. En vez de ello, quiero llamar tu atención sobre la cantidad de poder de cómputo que tenemos a disposición. Específicamente, cuánto no estamos aprovechando para hacer nuestras vidas como programadores un poco más fáciles. ¿Constantemente estás teniendo dificultades para tener suficiente poder de cómputo durante la jornada de trabajo? Si es así, ¿qué están haciendo tus servidores de prueba fuera de las horas de trabajo normal? A menudo están sin carga durante la noche y los fines de semana. Puedes usar eso a tu favor.

¿Te has sentido culpable de confirmar un cambio sin ejecutar todas las pruebas? Una de las razones principales de que los programadores no ejecutan los conjuntos de pruebas antes de hacer `commit` al código se debe a la cantidad de tiempo que puede tomar. Cuando las fechas límite se avecinan y la presión acecha, los humanos naturalmente empezamos a tomar atajos. Una forma de abordar esto es romper los largos conjuntos de pruebas en dos o más perfiles. Uno pequeño, un perfil de pruebas obligatorio que sea rápido de ejecutar, te ayudará a asegurarte de que las pruebas se ejecuten antes de cada commit. El total de lo perfiles (incluyendo el perfil obligatorio, sólo para estar seguros) puede ser automatizado para ejecutarse durante la noche, listo para reportar los resultados en la mañana.

¿Has tenido suficiente oportunidad de poner a prueba la estabilidad de tu producto? Las pruebas de más larga duración son vitales para identificar fugas de memoria y otros problemas de estabilidad. Rara vez se ejecutan durante el día, ya que consumen tiempo y recursos. Puedes automatizar una carga de prueba durante la noche y una un poco más larga durante el fin de semana. Del viernes 6.00 PM hasta las 6.00 AM del siguiente lunes hay 60 horas de tiempo potencial para las pruebas.

¿Estás obteniendo tiempo de calidad en tu entorno de pruebas de rendimiento? He visto altercados entre equipos para tener tiempo en estos entornos. En la mayoría de los casos ningún equipo obtiene tiempo de calidad durante el día, mientras que el ambiente está virtualmente inactivo durante las horas posteriores. Los servidores y la red no está ocupados durante la noche o los fines de semana. Es el momento ideal para ejecutar algunas pruebas de rendimiento de calidad.

¿Hay demasiadas permutaciones de pruebas manuales? En muchos casos tu producto está destinado a ser ejecutado en una variedad de plataformas. Por ejemplo, en 32 y 64 bits, en Linux, Solaris y Windows, o simplemente en diferentes versiones del mismo sistema operativo. Para empeorar las cosas, muchas aplicaciones modernas son expuestas a una plétora de mecanismos de transporte y protocolos (HTTP,

AMQP, SOAP, CORBA, etcétera). Probar manualmente todas estas permutaciones consume mucho tiempo y comúnmente se realizan cerca de una fecha de liberación debido a la presión de recursos. Por desgracia, puede ser demasiado tarde en el ciclo para capturar desagradables errores.

Las **pruebas automatizadas** que se ejecutan durante la noche o fin de semana asegurarán que todas estas permutaciones son puestas a prueba con mayor frecuencia. Con un poco de pensamiento y algo de conocimiento de secuencias de comandos (***scripting***) puedes programar unos cuantos trabajos cron para poner en marcha algunas pruebas durante la noche y los fines de semana. Hay también muchas herramientas de prueba por ahí que podrían ser útiles. Algunas organizaciones incluso tienen granjas de servidores que turnan servidores a través de diferentes departamentos y equipos para asegurar que los recursos son utilizados eficientemente. Si esto está disponible en tu empresa, puedes enviar las pruebas para que sean ejecutadas en la noche o durante los fines de semana.

Las pruebas son el rigor ingenieril del desarrollo de software

Por Neal Ford · Traducción: Espartaco Palma

Los desarrolladores aman usar metáforas torturadoras cuando se trata de explicar a los miembros de su familia, esposas y otros no técnicos qué es lo que hacen. Con frecuencia recurrimos a la construcción de puentes y otras disciplinas de ingenierías "duras". Todas estas metáforas caen rápidamente, sin embargo, cuando intentas presionar hacia a ellas demasiado duro. Resulta que el desarrollo de software no es como muchas de las otras disciplinas de la ingeniería, "duras" en muchos aspectos importantes.

Comparado con las ingenierías "duras", el mundo del desarrollo de software está en el mismo lugar donde los constructores de puentes estaban cuando la estrategia común era construir el puente y lanzar algo pesado sobre él. Si se mantenía de pie, era un buen puente. Si no, bueno, era tiempo de regresar a la mesa de dibujo. Durante los últimos miles de años, los ingenieros han desarrollado las matemáticas y física que usan para una solución estructural sin tener que construirlo para ver lo que hace. No tenemos nada como eso en el software, y quizás nunca lo tendremos, porque el software es, de hecho, algo muy diferente. Para una exploración profunda de la comparación entre "ingeniería" de software y la ingeniería normal, lee el libro "What's Software Design", escrito por Jack Reeves en C++ Journal en 1992, es un clásico. A pesar de que fue escrito hace casi dos décadas, es aún remarcablemente exacto. Él pintó un panorama sombrío en esta comparación, pero lo que faltaba en 1992 era una fuerte prueba ***Ethos*** para el software.

Probar cosas "duras" es difícil porque tienes que construirlo para probarlo, lo cual desalienta la construcción especulativa sólo para ver qué pasará. Pero el proceso de construcción de software es ridículamente barato. Hemos desarrollado todo un ecosistema de herramientas que hacen que sea fácil hacer precisamente eso: pruebas unitarias, objetos de imitación, arneses de pruebas y un montón de otras cosas. A otros ingenieros les encantaría ser capaces de hacer algo y probarlo bajo condiciones realistas. Como desarrolladores de software debemos abrazar las pruebas como la verificación primaria (pero no la única) para el software. En lugar de esperar por algún tipo de cálculo de software, ya tenemos las herramientas a nuestra disposición para asegurar buenas prácticas de ingeniería. Visto de esta manera, ahora tenemos municiones contra los directivos que dicen: "No tenemos tiempo para pruebas". Un constructor de puentes nunca escuchará de su jefe: "No te molestes en hacer el análisis estructural para esa construcción, tenemos un plazo muy corto". El reconocimiento de que la prueba es, de hecho, el camino para la reproducción y la calidad de software nos permite, como desarrolladores, regresar los argumentos contra su irresponsabilidad profesional.

Las pruebas toman su tiempo, al igual que el análisis estructural lleva su tiempo. Ambas actividades garantizan la calidad del producto final. Es hora de que los desarrolladores tomen el mando de la

responsabilidad de lo que producen. Las pruebas por sí mismas no son suficientes, pero son necesarias. Probar es el rigor ingenieril del desarrollo de software.

Pensando en estados

Por Niclas Nilsson · Traducción: Espartaco Palma

La gente en el mundo real tiene una rara relación con los estados. Esta mañana me paré en la tienda local preparándome para otro día de convertir cafeína en código. Debido a que mi forma favorita de hacerlo es tomando un latte, al no encontrar leche me dijo la empleada:

—Disculpa, estamos super-dupe mega faltos de leche.

Para un programador, eso es una sentencia rara. Puedes tener leche o no. No hay escalas cuando se trata de estar sin el lácteo. Quizás ella estaba tratando de decirme que les faltará leche por una semana, pero el resultado era el mismo: día de espresso para mí.

En muchas situaciones del mundo real, la actitud relajada de la gente con los estados no es un problema. Sin embargo, desafortunadamente, muchos programadores son también algo despistados con respecto a los estados y eso sí es un problema.

Considere una tienda de ventas en línea que sólo acepta tarjetas de crédito y que no factura a los clientes, tiene una clase `Orden` conteniendo este método:

EJEMPLO DE CÓDIGO

```
public boolean isComplete() {
    return isPaid() && hasShipped();
}
```

Razonable, ¿no es así? Bueno, incluso si la expresión es amablemente extraída en un método en vez de copiar-pegar en todos lados, la expresión no debería existir del todo. El hecho es que sí resalta un problema. ¿Por qué? La orden no puede ser enviada antes de que sea pagada. Por lo tanto, `hasShipped` no puede ser verdadero a menos que `isPaid` sea verdadero, lo cual hace parte de la expresión redundante. Por cuestiones de claridad puedes querer aún el `isComplete` en el código, pero entonces debería verse como esto:

EJEMPLO DE CÓDIGO

```
public boolean isComplete() {
    return hasShipped();
}
```

En mi trabajo veo todo el tiempo ambas: revisiones faltantes y revisiones redundantes. Este ejemplo es pequeño, pero cuando agregas cancelaciones y reembolso, esto se vuelve más complejo y la necesidad

de un buen manejo de estados se incrementa. En este caso, una orden puede estar sólo en uno de tres distintos estados:

En progreso: puede agregar o remover elementos. No se puede enviar.

Pagado: no puede agregar o remover elementos. Puede ser enviado.

Enviado: Terminado. No se aceptan más cambios.

Estos estados son importantes y necesitas revisar que estés en el estado esperado antes de realizar operaciones, y que tú sólo puedas moverte a un estado legal desde donde estás. En resumen, tienes que proteger tus objetos cuidadosamente, en los lugares correctos.

Sin embargo, ¿cómo empezar a pensar en estados? Extrayendo expresiones significativas a los métodos es un buen inicio, pero es sólo un comienzo. Las bases están en entender las máquinas de estados. Se que puedes tener malos recuerdos de tus clases de Ciencias Computacionales, pero déjalos atrás. Las máquinas de estados no son especialmente difíciles. Visualízalas para hacerlas simples de entender y fáciles de hablar de ellas. Haz tu código Test-drive para desentrañar los estados válidos e inválidos, las transiciones y mantenlas correctas. Estudia el patrón State. Cuando te sientas cómodo, lee sobre Diseño por Contrato. Esto ayuda a asegurarte un estado apropiado al validar los datos de entrada y los objetos por sí mismos al entrar y salir de cada método público.

Si tu estado no es correcto, hay un bug y estás en riesgo de tirar a la basura datos si no abortas. Si encuentras que las revisiones de estado son ruidosas, aprende cómo usar una herramienta de generación de código, ***weaving*** o aspectos para ocultarlos. Independientemente del enfoque que elijas, pensar en estados hará que tu código sea más simple y más robusto.

Dos cabezas son a menudo mejores que una

Por Adrian Wible · Traducción: Espartaco Palma

La programación requiere pensamiento profundo, y los pensamientos profundos requieren soledad. Así va el estereotipo del programador.

Este enfoque de "lobo solitario" de la programación está dando paso a un enfoque colaborativo, el cuál, puedo decir, mejora la calidad, productividad y satisfacción laboral de los programadores. Este enfoque tiene a los desarrolladores trabajando más cerca entre sí y también con los no desarrolladores –analistas de negocio y sistemas, profesionales de control de calidad y usuarios–.

¿Qué significa esto para los desarrolladores? Ser el experto técnico ya no es suficiente. Debes ser más efectivo trabajando con otros.

La colaboración no se trata de preguntar y responder, o sentarse en reuniones. Se trata de arremangarse con alguien más para atacar conjuntamente el trabajo.

Soy un gran admirador de la programación en pareja. Puedes llamar a esto "colaboración extrema". Como desarrollador, mis habilidades crecen cuando hago pareja. Si soy más débil que mi compañero en el dominio o tecnología, claramente aprendo de su experiencia. Cuando soy más fuerte en algún aspecto, aprendo más sobre lo que conozco y no conozco al tener que explicarme. Invariablemente, ambos traemos algo a la mesa y aprendemos mutuamente.

En pareja, cada uno de nosotros llevamos nuestras experiencias de programación colectiva –tanto de dominio como técnica– al problema en cuestión y podemos aportar agudeza y experiencias únicas al escribir software efectiva y eficientemente. Incluso en caso de desequilibrio extremo en el dominio o conocimiento técnico, el participante más experimentado invariablemente aprende algo del otro –quizás un nuevo atajo del teclado o exposición a una nueva herramienta o biblioteca–. Para los miembros menos experimentados del par, es una gran manera de ponerse al día.

La programación en pareja es popular con, pero no exclusivamente a, promotores del desarrollo ágil del software. Alguien que se opone a la pareja sugiere: "¿porqué debería pagar a dos programadores para hacer el trabajo de uno?". Mi respuesta es que, en efecto, no debería. Argumento que el emparejamiento incrementa la calidad, el entendimiento del dominio y tecnología, técnicas (como los trucos del `IDE`) y mitiga el impacto del riesgo de lotería (uno de tus desarrolladores expertos se gana la lotería y renuncia al siguiente día).

¿Cuál es el valor a largo plazo de aprender un nuevo atajo del teclado? ¿Cómo medimos la mejora global de la calidad del producto resultante del emparejamiento? ¿Cómo medimos el impacto de que tu compañero no te permita adoptar un enfoque sin salida en la solución de un problema difícil? Un estudio cita un incremento del 40% en eficacia y velocidad (J T Nosek, "The Case for Collaborative Programming", Communications of the

ACM, Marzo de 1998). ¿Cuál es el valor de mitigar tu “lotería de riesgo”? Muchas de estas ganancias son difíciles de medir.

¿Quién debería hacer pareja con quién? Si eres nuevo, es importante encontrar un miembro del equipo que tenga conocimientos. Tan importante como encontrar quien tenga buenas habilidades interpersonales y de entrenador. Si no tienes mucha experiencia del dominio, emparéjate con un experto.

Si no estás convencido, experimenta: colabora con tus colegas. Haz pareja en un problema retorcido e interesante. Ve cómo se siente. Inténtalo unas cuantas veces.

CONSEJO 86 DE 97

Dos fallos pueden hacer un acierto (y es difícil de arreglar)

Por Allan Kelly · Traducción: Espartaco Palma

El código nunca miente, pero puede contradecirse. Algunas contradicciones llevan a esos momentos de: “¿cómo es posible que esto funcione?”.

En una entrevista, el diseñador principal del software del módulo lunar Apolo 11, Allan Klumpp, reveló que el software que controlaba los motores tenía un error que hacía el módulo de aterrizaje inestable. Sin embargo, otro error fue compensado por el primero y el software fue usado por los aterrizajes lunares del Apolo 11 y 12 antes de que el error fuera encontrado y arreglado.

Considera una función que retorna un estatus de finalización. Imagina que retorna `false` cuando debería regresar un `true`. Ahora imagina que la llamada de función olvida comprobar el valor de retorno. Todo funciona bien hasta que un día alguien nota la falta de verificación y la inserta.

O considera una aplicación que almacena su estado en un documento `XML`. Imagina que uno de los nodos está escrito incorrectamente como “TimeToLive” en vez de “TimeToDie”, como la documentación dice que debería. Todo parece estar bien mientras el código de escritura y el código de lectura contienen ambos el mismo error. Pero arregla uno, o agrega una nueva aplicación de lectura del mismo documento, y la simetría se rompe, al igual que el código.

Cuando dos defectos en el código crean un defecto visible, el enfoque metodológico para arreglar la falla puede, por sí mismo, romperlo. El desarrollador recibe un reporte de error, encuentra el defecto, lo arregla y lo vuelve a probar. Sin embargo, el fallo reportado aún ocurre, debido a que un segundo defecto está en funcionamiento. Así que el primer arreglo se quita, el código es inspeccionado hasta que el segundo defecto es encontrado, y un arreglo se aplica. Pero el primer defecto ha regresado, el fallo reportado aún se ve, así que se deshace el segundo arreglo. El proceso se repite, pero ahora el desarrollador ha desestimado dos posibles soluciones y está buscando una tercera, que nunca va a funcionar.

La interacción entre dos defectos de código que aparecen como un defecto visible no sólo hace difícil arreglar el problema, además, lleva a los desarrolladores a callejones sin salida, sólo para descubrir que intentaron la respuesta correcta desde el inicio.

Esto no pasa sólo en el código: el problema también existe en los documentos de requerimientos escritos. Y puede extenderse, viralmente, de un lugar a otro. Un error en el código compensa un error en la descripción escrita.

Puede extenderse a la gente también: los usuarios aprenden que cuando la aplicación dice “Izquierda” se refiere a la “Derecha”, así que ajustan su comportamiento, incluso lo pasan al nuevo usuario: “recuerda que la aplicación dice que hagas clic al botón izquierdo cuando realmente se refiere al botón derecho”. Arregla ese error y, de repente, los usuarios necesitan reentrenamiento.

Fallos sencillos pueden ser fáciles de ver y de arreglar. Son los problemas con múltiples causas, que necesitan múltiples cambios, los que son difíciles de resolver. En parte es porque los problemas fáciles tienden a ser arreglarlos con relativa rapidez y se quedan los más difíciles para una fecha posterior.

No hay un consejo simple que se pueda dar en cómo localizar fallos surgidos de defectos simpatéticos. Es necesario darse cuenta de la posibilidad, una cabeza clara y voluntad de considerar todas las posibilidades.

Codificación Ubuntu para tus amigos

Por Aslam Khan · Traducción: Espartaco Palma

A menudo escribimos código en el aislamiento y refleja nuestra interpretación personal de un problema, así como una solución personalizada. Podemos ser parte de un equipo y aun así estar aislados. Olvidamos todo tan fácilmente que este código creado en el aislamiento será ejecutado, usado, extendido y ha confiado a otros. Es fácil pasar por alto el aspecto social de la creación de software. Crear software es un ejercicio técnico mezclado con un ejercicio social. Sólo necesitamos levantar nuestra cabeza para darnos cuenta de que no estamos trabajado aisladamente y tenemos responsabilidades compartidas con respecto a incrementar la probabilidad de éxito de todos, no sólo del equipo de desarrollo.

Podemos escribir código de buena calidad en el aislamiento, mientras nos perdemos en nosotros mismos. Desde alguna perspectiva, eso es un enfoque egocéntrico (no ego como en arrogante, sino ego como en lo personal). También es una visión Zen y es sobre ti, en ese momento de la creación de código. Siempre intento vivir en el momento porque ayuda a estar más cerca de la calidad, pero entonces vivo en mi momento. ¿Qué pasa con el momento de mi equipo? ¿Es mi momento el mismo que el del equipo?

En Zulu, la filosofía de Ubuntu se resume en “Umuntu ngumuntu ngabantu”, que se podría traducir como “una persona es una persona a través de (otras) personas”. Me siento mejor porque tú me haces mejor a través de tus buenas acciones. La otra cara es que eres peor en lo que haces cuando soy malo en lo que hago. Entre desarrolladores, podemos reducirlo a “un desarrollador es un desarrollador a través de (otros) desarrolladores”. Si lo llevamos hasta el metal, entonces “el código es código a través de código (de los otros)”.

La calidad del código que escribo afecta la calidad del código que tu escribes. ¿Qué pasa si mi código es de baja calidad? Incluso si escribes un código muy limpio, los puntos donde usas mi código es donde la calidad de tu código se degrada. Puedes aplicar muchos patrones y técnicas para limitar el daño, pero el daño ya está hecho. He causado que tú hagas más de lo que necesitas hacer simplemente porque no pensé en ti cuando estaba viviendo mi momento.

Puede que considere mi código como limpio, pero puedo aún hacerlo mejor sólo codificando Ubuntu. ¿A que se parece el código Ubuntu? Se ve como un buen **código limpio**. No se trata del código, el artefacto. Se trata del acto de crear ese artefacto. Codificar para tus amigos con Ubuntu ayudará a que tu equipo viva tus valores y refuerce sus principios. La siguiente persona que toque tu código, en cualquier forma, será una mejor persona y un mejor desarrollador.

El Zen se trata de lo individual. Ubuntu es acerca del Zen para un grupo de personas. Muy, muy raramente creamos código para nosotros mismos.

Las herramientas Unix son tus amigas

Por Diomidis Spinellis · Traducción: Espartaco Palma

Si en mi camino al exilio en una isla desierta tuviera que escoger entre un `IDE` y un conjunto de herramientas Unix, yo escogería las herramientas Unix sin pensarlo dos veces. Aquí están las razones por las cuáles deberías dominar las herramientas Unix.

Primero, los IDE se enfocan en lenguajes específicos, mientras las herramientas Unix pueden trabajar con cualquier cosa que aparezca en modo textual. En los ambientes de desarrollo de hoy en día, donde los nuevos lenguajes y notaciones florecen cada año, aprender a trabajar de la forma Unix es una inversión que se pagará con el tiempo una y otra vez.

Además, mientras los IDE ofrecen sólo los comandos que sus desarrolladores concibieron, con las herramientas Unix puedes realizar cualquier tarea imaginable. Piensa en ello como (los clásico pre- Biónico) bloques Lego: creas tus propios comandos combinando las pequeñas pero versátiles herramientas Unix. Por ejemplo, la siguiente secuencia es una implementación basada en texto del análisis de firmas de Cunningam; una secuencia de cada punto y coma, llaves y comillas que puede revelar mucho sobre el contenido del archivo:

EJEMPLO DE CÓDIGO

```
for i in *.java; do echo -n "$i: " sed 's/[^"{};]//g' $i | tr -d
'\n' echo done
```

En suma, cada operación del IDE que aprendes es específica a esa tarea; por ejemplo, agregar un nuevo paso de depuración en la configuración de construcción del proyecto. En contraste, afilar tus habilidades con las herramientas Unix te hace más efectivo en cualquier tarea. Como un ejemplo, he empleado la herramienta sed en la secuencia de comandos precedentes para modificar la construcción de un proyecto para la compilación cruzada en múltiples arquitecturas de procesador.

Las herramientas Unix fueron desarrolladas en una época en la que una computadora multiusuario tenía 128kB de RAM. El ingenio que tuvo su diseño significa que en estos días pueden manejar enormes conjuntos de datos con extremada eficiencia. La mayoría de las herramientas trabajan como filtros, procesando sólo una línea a la vez, significando que no hay límite superior en la cantidad de datos que pueden manejar. ¿Quieres buscar un número de ediciones almacenadas en medio terabyte del respaldo de la Wikipedia en inglés? La simple invocación de

EJEMPLO DE CÓDIGO

```
grep '<revision>' | wc –l
```

te dará la respuesta sin siquiera sudar. Si encuentras una secuencia de comandos útil, puedes empacarla fácilmente en un script de shell, usando algunos poderosos constructos de programación, tales como hacer piping de datos en ciclos y condicionales. Más impresionante aún, los comandos Unix ejecutados como ***pipelines***, como el arriba descrito, distribuirá su carga con naturalidad a través de las muchas unidades de procesamiento de los CPU multi-core modernos.

Su génesis en “pequeño es bello” y las implementaciones de software libre de las herramientas Unix las hacen disponibles ubicuamente, incluso en plataformas de recursos restringidos, como mi reproductor multimedia de la sala o el router `DSL`. Es poco probable que tales dispositivos ofrezcan una poderosa interface gráfica, pero frecuentemente incluyen la aplicación BusyBox, la cual provee la mayoría de las herramientas comúnmente usadas. Y si estás desarrollando en Windows, el ambiente cygwin te ofrece todas las herramientas Unix imaginables, en forma de ejecutable y código fuente.

Por último, si ninguna de las herramientas disponibles se adecua a tus necesidades, es muy fácil extender el mundo de las herramientas Unix. Sólo escribe un programa (en cualquier lenguaje que elijas) que juegue con unas pocas y sencillas reglas: tu programa debe realizar sólo una tarea sencilla; debe leer datos como líneas de texto de su entrada estándar y debe mostrar los resultados sin adornos, encabezados ni otros ruidos en su salida estándar. Los parámetros que afectan la operación de la herramienta se dan en la línea de comandos. Sigue estas reglas y “tuya será la Tierra y todo lo que hay en ella”.

Usa el algoritmo y estructura de datos correctos

Por JC van Winkel · Traducción: Espartaco Palma

Un gran banco con muchas sucursales se quejó de que las nuevas computadoras que había comprado para los cajeros eran muy lentas. Esto era antes de que todos usaran la banca electrónica y los cajeros automáticos no estaban tan extendidos como lo están ahora. La gente visitaba el banco mucho más frecuentemente y se hacían largas filas debido a las computadoras lentas. En consecuencia, el banco amenazó con romper su contrato con el proveedor.

El proveedor envió un especialista en análisis y ***tuning*** para determinar la causa de los retrasos. Pronto encontró un programa específico ejecutándose en la terminal consumiendo casi toda la capacidad del CPU. Usando una herramienta de perfilado se enfocó en el programa y pudo ver la función culpable. El código se leía:

EJEMPLO DE CÓDIGO

```
for (i=0; i<strlen(s); ++i) {
  if (... s[i] ...) ...
}
```

La cadena `s` tenía, en promedio, miles de caracteres de longitud. El código (escrito por el banco) fue rápidamente cambiado y los cajeros vivieron felices por siempre…

¿No debía el programador haberlo hecho mejor que un código que innecesariamente escalaba cuadráticamente?

Cada llamada a `strlen` recorría cada uno de los miles de caracteres en la cadena para encontrar su carácter de terminación nula. La cadena, sin embargo, nunca cambiaba. Al determinar su longitud por adelantado, el programador podía haber ahorrado cientos de llamadas a `strlen` (y millones de ejecuciones del bucle):

EJEMPLO DE CÓDIGO

```
n=strlen(s);
for (i=0; i<n; ++i) {
  if (... s[i] ...) ...

}
```

Todos conocen el viejo dicho “primero haz que funcione, luego haz que funcione rápido” para evitar las trampas de la micro-optimización. Pero el ejemplo de arriba casi nos hace creer que el programador siguió el maquiavélico adagio “primero haz que funcione lentamente”.

Este tipo de descuido es algo con lo podrías cruzarte más de una vez. Y no es sólo un “no reinventes la rueda”. Algunas veces los programadores novatos sólo empiezan a escribir sin realmente pensar y de repente han “inventado” el ordenamiento por burbuja. Incluso podrían estar alardeando sobre eso.

El otro lado de elegir el algoritmo correcto es la elección de la estructura de datos. Puede hacer una gran diferencia: usar una lista enlazada para una colección de millones de elementos por las que quieres buscar –comparada con una estructura de datos de hash– va a tener un gran impacto en la apreciación del usuario de tu programación.

Los programadores no deberían reinventar la rueda y deberían usar bibliotecas existente cuando fuera posible. Pero, para ser capaces de evitar problemas como el del banco, deberían también ser educados acerca de los algoritmos y cómo escalan. ¿Es sólo la vistosidad en los editores lo que hace que sean tan lentos como los anticuados programas como WordStar en la década de los ochenta? Muchos dicen que el reúso en la programación es de gran importancia. Por encima de todo, sin embargo, los programadores deben saber cuándo, qué y cómo reutilizar. Para poder hacer eso deben tener el dominio del problema y los algoritmos y estructuras de datos.

Un buen programador debería también saber cuándo usar un algoritmo abominable. Por ejemplo, si el dominio del problema dicta que nunca puede haber más de cinco elementos (como el número del dado en el juego Yahtzee) y sabes que siempre tendrás que ordenar, al menos, cinco elementos. En este caso, el ordenamiento por burbuja puede ser la más eficiente forma de ordenar los elementos. Cada perro tiene su día.

Entonces, lee algunos buenos libros y asegúrate de que los entiendas. Si realmente lees bien El arte de la programación, de Donald Knuth, podrías incluso ser afortunado: encuentra una equivocación del autor y gana uno de los cheques de dólares hexadecimales ($2.56).

CONSEJO 90 DE 97

Los registros detallados perturbarán tu sueño

Por Johannes Brodwall · Traducción: Espartaco Palma

Cuando me encuentro un sistema que ya ha estado en desarrollo o producción por un tiempo, la primera señal de un verdadero problema es siempre un registro sucio. Sabes a lo que me refiero. Cuando al hacer clic en un link de flujo normal de una página web, resulta en un diluvio de mensajes en el único registro que el sistema provee. Demasiados registros pueden ser inútiles como ninguno.

Si tus sistemas son como los míos, cuando se termina tu trabajo empieza el trabajo de alguien más. Después de que el sistema ha sido desarrollado, es de esperar que vivirá una larga y próspera vida de servicio a los clientes. Si tienes suerte. ¿Cómo sabrás si algo va mal cuando el sistema está en producción y cómo lidiar con él?

Quizás alguien más lo monitoreará por ti o lo monitorearás tú mismo. De cualquier forma, los registros probablemente serán parte del monitoreo. Si algo sucede y tienes que estar despierto para lidiar con él, entonces quieres estar seguro que hay una buena razón en ello. Si el sistema está muriendo, quiero saberlo. Pero si es sólo hipo, preferiría disfrutar de mi bello sueño.

Para muchos sistemas, el primer indicador de que algo está mal es un mensaje de registro escrito en alguna bitácora. Generalmente, éste será un registro de error. Así que hazte un favor: asegúrate desde el día uno de que si registras algo en la bitácora de errores, estás dispuesto a tener a alguien llamando y despertándote a la mitad de la noche por ello. Si puedes simular carga en tu sistema durante las pruebas, mirar en una bitácora de errores libre de ruido es también una buena primera indicación de que tu sistema es razonablemente robusto. O una alerta temprana si no lo es.

Los sistemas distribuidos agregan otro nivel de complejidad. Tienes que decidir cómo hacer frente a uno de dependencia externa. Si tu sistema está muy distribuido, esto será una ocurrencia común. Asegúrate de que tu política de registro lo tome en cuenta.

En general, la mejor señal de que todo está bien es que los mensajes de menor prioridad están tildando felizmente. Deseo algo así como un mensaje de registro de nivel INFO por cada evento importante de la aplicación.

Una bitácora desordenada es un indicador de que el sistema será difícil de controlar una vez que llegue a producción. Si no esperas que nada se muestre en la bitácora de error, será mucho más fácil saber qué hacer cuando algo aparezca.

El WET dispersa los cuellos de botella en el rendimiento

Por Kirk Pepperdine · Traducción: Espartaco Palma

La importancia del **principio DRY** (No te repitas) es que codifica la idea de que cada pieza del conocimiento en un sistema debería tener una representación única. En otras palabras, el conocimiento debería estar contenido en una implementación única. La antítesis de DRY es WET (Write Every Time, escríbelo todas las veces). Nuestro código es WET cuando el conocimiento es codificado en varias distintas implementaciones. Las implicaciones de rendimiento de DRY versus WET quedan claras cuando consideras los numerosos efectos en un perfil de rendimiento.

Comenzamos considerando una característica en nuestro sistema, digamos X, que es un cuello de botella de CPU. Digamos que la característica X consume el 30% del CPU. Ahora digamos que la característica X tiene diez diferentes implementaciones. En promedio, cada implementación consume 3% del CPU. En este nivel de uso de CPU no es útil preocuparse si estamos buscando una victoria rápida, es común que olvidemos que esta característica es nuestro cuello de botella. Sin embargo, digamos que, de alguna manera, reconocimos la característica X como un cuello de botella. Ahora estamos con el problema de encontrar un arreglo en cada implementación. Con WET tenemos diez diferentes implementaciones que necesitamos buscar y reparar. Con DRY veríamos claramente el 30% de uso de CPU y tendríamos una décima parte de código que arreglar. ¿Mencioné que no tenemos tiempo que perder buscando cada implementación?

Hay un caso de uso en el cual frecuentemente nos sentimos culpables de violar el principio DRY: nuestro uso de colecciones. Una técnica común de implementar una consulta sería el iterar sobre una colección y entonces aplicar la consulta para cada elemento:

EJEMPLO DE CÓDIGO

```
public class UsageExample {
    private ArrayList<Customer> allCustomers = new ArrayList<Customer>();
    // ...
    public ArrayList<Customer> findCustomersThatSpendAtLeast(Money amount) {
        ArrayList<Customer> customersOfInterest = new ArrayList<Customer>();
        for (Customer customer: allCustomers) {
            if (customer.spendsAtLeast(amount))
               customersOfInterest.add(customer);
        }
        return customersOfInterest;
    }
}
```

Al exponer esta colección en bruto a los clientes, hemos violado la encapsulación. Esto no sólo limita nuestra habilidad para **refactorizar**, obliga a los usuarios de nuestro código a violar el principio DRY al tener cada uno de ellos que reimplementar potencialmente la misma consulta. Esta situación se puede evitar fácilmente al quitar la colección en bruto del `API`. En este ejemplo podemos introducir un nuevo tipo de colección de dominio específico llamado `CustomerList`. Esta nueva clase es más semántica en la línea de nuestro dominio. Actuará como una casa natural para todas nuestras consultas.

Tener esta nueva colección nos permitirá ver de forma sencilla si esta consulta es un cuello de botella en el rendimiento. Al incorporar las consultas en la clase eliminamos la necesidad de exponer las elecciones de representación, tales como `Arraylist`, a nuestros clientes. Esto nos da la libertad de alterar esta implementación sin el miedo de violar los contratos de los clientes:

EJEMPLO DE CÓDIGO

```
public class CustomerList {
    private ArrayList<Customer> customers = new ArrayList<Customer>();
    private SortedList<Customer> customersSortedBySpendingLevel = new SortedL
    // ...
    public CustomerList findCustomersThatSpendAtLeast(Money amount) {
        return new CustomerList(customersSortedBySpendingLevel.elementsLarger
    }
}

public class UsageExample {
    public static void main(String[] args) {
        CustomerList customers = new CustomerList();
        // ...
        CustomerList customersOfInterest = customers.findCustomersThatSpendAt
        // ...
    }
}
```

En este ejemplo, la adherencia a DRY nos permite introducir un esquema de índice alterno con `SortedList` usando una llave en el nivel de gasto de nuestros clientes. Más importante que los detalles específicos de este ejemplo, en particular, seguir el principio DRY nos ayuda a encontrar y reparar cuellos de botella en el rendimiento que habrían sido más difíciles de encontrar si el código fuera WET.

Cuando Programadores y Testers colaboran

Por Janet Gregory · Traducción: Espartaco Palma

Algo mágico sucede cuando los testers y programadores empiezan a colaborar. Hay menos tiempo perdido mandando bugs de ida y de regreso a través del sistema de rastreo de defectos. Menos tiempo se desperdicia intentando imaginar si algo es realmente un error o una nueva característica, y más tiempo es usado desarrollando buen software para satisfacer las expectativas de los clientes. Hay muchas oportunidades para comenzar a colaborar, incluso antes de que la codificación inicie.

Los testers pueden ayudar a los clientes a escribir y automatizar las pruebas de aceptación usando el lenguaje de su dominio con herramientas tales como ***Fit*** (***Framework for Integrated Test***). Cuando estas prueban son entregadas a los programadores antes de que la codificación inicie, el equipo está practicando el Desarrollo Conducido por Pruebas de Aceptación (***Acceptance Test Driven Development***, ATDD). Los programadores escriben sus arreglos para ejecutar las pruebas, y entonces codifican para hacer que las pruebas pasen. Estas pruebas se convierten en parte de la suite de regresión. Cuando esta colaboración ocurre, las pruebas funcionales se completan de manera temprana, lo que da tiempo para las pruebas exploratorias en condiciones extremas o a través de flujos de trabajo con un rango más amplio.

Podemos dar un paso más adelante. Como tester puedo suministrar la mayoría de mis ideas de prueba antes de que los programadores codifiquen una nueva característica. Cuando le pregunto a los programadores si tienen alguna sugerencia, ellos casi siempre me proveen la información que me ayuda con una mejor cobertura de pruebas, o me ayuda a evitar gastar mucho tiempo en pruebas innecesarias. Frecuentemente hemos prevenido defectos porque las pruebas clarifican muchas de las ideas iniciales. Por ejemplo, en un proyecto en el que estaba, la prueba ***Fit*** que le di al programado mostraba los resultados esperados de una consulta que respondía a una búsqueda con comodines. El programador pretendía codificar sólo búsquedas de palabras completas. Pudimos hablar con el cliente y determinar la interpretación correcta antes de que la codificación iniciara. Al colaborar prevenimos el defecto, lo cual nos ahorró a ambos un montón de tiempo.

Los programadores pueden colaborar con los testers para crear también una automatización exitosa. Ellos entienden las buenas prácticas de codificación y pueden ayudar a los testers a configurar una robusta ***suite*** de automatización de pruebas que funcione para todo el equipo. Muchas veces he visto proyectos de automatización que fallan porque las pruebas están mal diseñadas. Las pruebas intentan probar mucho o los testers no han entendido lo suficiente acerca de la tecnología para ser capaces de mantener las pruebas independientes. Los testers son frecuentemente el cuello de botella, así que tiene sentido para los programadores el trabajar con ellos en las tareas como la automatización. Al trabajar con los testers para entender qué puede ser probado tempranamente, quizás al proporcionar una herramienta sencilla, dará a los programadores otro ciclo de retroalimentación que les ayudará, a largo plazo, a entregar mejor código.

Cuando los testers dejan de pensar que su único trabajo es romper el software y buscar errores en el código de los programadores, los programadores dejan de pensar que los testers “van por ellos” y están más abiertos a la colaboración. Cuando los programadores empiezan a darse cuenta de que son responsables de construir calidad dentro de su código, el realizar pruebas es algo natural para el producto y el equipo puede automatizar más pruebas de regresión juntos. La magia del trabajo en equipo comienza.

Escribe código como si tuvieras que mantenerlo por el resto de tu vida

Por Yuriy Zubarev · Traducción: Espartaco Palma

Puedes preguntarle a 97 personas lo que todo programador debería saber y hacer, y podrás escuchar 97 respuestas distintas. Esto podría ser abrumador e intimidante al mismo tiempo. Todo consejo es bueno, todos los principios son sólidos y todas las historias son convincentes, pero ¿por dónde empezar? Más importante aún, una vez que has comenzado, ¿cómo te mantienes al día con todas las mejores prácticas que has aprendido para hacer de ellas una parte integral de tus prácticas de programación?

Creo que la respuesta reside en tu estado de ánimo o, más claramente, en tu actitud. Si no te preocupas por tus compañeros desarrolladores, ***testers***, administradores, personal de venta y mercadotecnia, así como los usuarios finales, entonces no estarás dispuesto a emplear el **Desarrollo basado en Pruebas** (***Test-Driven Development***) o escribir comentarios claros en tu código, por ejemplo. Hay una manera sencilla de ajustar tu actitud y siempre estar dispuesto a entregar productos de la mejor calidad::

> IDEA CLAVE
>
> Escribe código como si tuvieras que mantenerlo por el *resto de tu vida*.

Eso es todo. Si aceptas esta idea, sucederán muchas cosas maravillosas. Si vas aceptar que ninguno de tus empleadores previos o actuales tiene derecho a llamarte a la mitad de la noche pidiéndote que expliques las decisiones que tomaste mientras escribías el método `fooBar`, entonces deberías mejorar gradualmente para convertirte en un programador experto. Naturalmente querrías llegar a mejores nombres de variables y métodos. Te alejarías de bloques de código que contienen cientos de líneas. Buscarías, aprenderías y usarías **patrones de diseño**. Escribirías comentarios, probarías tu código y refactorarías continuamente. Mantener todo el código que has escrito por el resto de tu vida será también un esfuerzo escalable. Por lo tanto, no tendrías más opción que convertirte en alguien mejor, más listo y más eficiente.

Si lo reflexionas, el código que escribiste hace muchos años todavía influye en tu carrera, te guste o no. Dejas un rastro de tu conocimiento, actitud, tenacidad, profesionalismo, nivel de compromiso y grado de disfrute con cada método, clase y módulo que diseñas y escribes. La gente se formará opiniones de ti con base en el código que ven. Si esas opiniones son constantemente negativas, entonces obtendrás menos de tu carrera de lo que esperabas. Preocúpate por tu carrera, tus clientes y todos los usuarios con cada línea de código; escribe código como si tuvieras que mantenerlo por el resto de tu vida.

Escribe pequeñas funciones usando ejemplos

Por Keith Braithwaite · Traducción: Espartaco Palma

Nos gustaría escribir código que fuese correcto y tener evidencia en mano de que es correcto. Puede ayudar con ambos temas pensar en el “tamaño” de una función. No en el sentido de la cantidad de código que implementa una función, a pesar de que es interesante; sino, más bien, del tamaño como una función matemática que nuestro código manifiesta.

Por ejemplo, en el juego de Go hay una condición llamada ***atari***, en la cual la piedra del jugador puede ser capturada por su oponente: una piedra con dos o más espacios libres adyacentes a él (llamados ***liberties***) no está en ***atari***. Puede ser difícil de contar cuántas ***liberties*** tiene una piedra, pero determinar el ***atari*** es fácil si se sabe. Podrías empezar escribiendo una función como esta:

EJEMPLO DE CÓDIGO

```
boolean atari(int libertyCount) libertyCount < 2
```

Esto es más grande de lo que parece. Una función matemática puede ser entendida como un conjunto, algún subconjunto del producto Cartesiano del conjunto que son su dominio (en este caso, un entero) y rango (en este caso, un booleano). Si esos conjuntos de valores fueran del mismo tamaño, como en Java, entonces sería `2L*(Integer.MAX_VALUE+(-1L*Integer.MIN_VALUE)+1L)` o `8,589,934,592` miembros en el conjunto int × boolean. La mitad son miembros de un conjunto que es nuestra función, así que para proveer una evidencia completa de que nuestra función es correcta necesitaríamos revisar al rededor de 4.3×10^9 ejemplos.

Ésta es la esencia de la afirmación de que las pruebas no pueden probar la ausencia de errores. Sin embargo, las pruebas pueden demostrar la presencia de características. Pero aún tenemos este tema del tamaño.

El dominio del problema nos ayuda. La naturaleza de Go significa que el número de ***liberties*** de una piedra no es cualquier entero, pero exactamente uno de `{1,2,3,4}`. Así pues, podríamos escribir alternativamente:

EJEMPLO DE CÓDIGO

```
LibertyCount = {1,2,3,4} boolean atari(LibertyCount libertyCount)
libertyCount == 1
```

Esto es mucho más manejable: la función calculada es ahora un conjunto con, cuando mucho, ocho miembros. De hecho, cuatro ejemplos seleccionados constituirían la evidencia de la certeza completa de que la función es correcta. Ésta es la razón por la cual es una buena idea usar tipos estrechamente relacionados al dominio del problema para escribir programas, en vez de tipos nativos. Usar tipos inspirados en dominios a menudo puede hacer que nuestras funciones sean mucho más pequeñas. Una forma de encontrar qué tipo sería es encontrar los ejemplos para comprobar en términos del dominio del problema, antes de escribir la función.

Escribe las pruebas para las personas

Por Gerard Meszaros · Traducción: Espartaco Palma

Estás escribiendo **pruebas automatizadas** para una parte o todo tu código de producción. ¡Felicidades! ¿Estás escribiendo tus pruebas antes de que escribas el código? ¡¡Mucho mejor!! El sólo hacerlo te convierte en uno de los primeros adoptantes de las más avanzadas prácticas de la ingeniería de software. Pero, ¿estás escribiendo buenas pruebas? ¿Cómo saberlo? Una manera es preguntar: "¿para quién estoy escribiendo estas pruebas?". Si la respuesta es "para mí, para ahorrarme el esfuerzo de corregir errores" o "para el compilador, con eso puede ser ejecutado", entonces las apuestas están en que no estás escribiendo las mejores pruebas posibles. Así que, ¿para quién deberías estar escribiendo las pruebas? Para las personas que tratan de entender tu código.

Las buenas pruebas actúan como documentación para el código que estás probando. Describen cómo funciona el código. Por cada escenario de uso la(s) prueba(s): Describe el contexto, un punto inicial o precondiciones que deben ser satisfechas; ilustra cómo el software es invocado; describe los resultados esperados o poscondiciones a ser verificadas.

Los diferentes escenarios de uso tendrán una versión distinta de cada una de ellas. Las personas que tratan de entender tu código deberían poder mirar unas cuantas pruebas y, al comparar estas tres partes de las pruebas en cuestión, ver qué causa que el código se comporte diferente. Cada prueba debería ilustrar claramente la relación de causa y efecto entre estas tres partes. Esto implica que lo que no es visible en las pruebas es tan importante como lo que es visible. Mucho código en las pruebas distrae al lector con trivialidades sin importancia. Cuando sea posible oculta dichas trivialidades detrás de llamados a métodos con significado; la **refactorización** "Extraer Método" es tu mejor amigo. Y asegúrate de darle a cada prueba un nombre con significado que describa el escenario de uso particular, con esto el lector de la prueba no tiene que hacer ingeniería inversa de cada prueba para entender de qué se tratan los distintos escenarios. Entre ellos, el nombre de las clases de prueba y los métodos de clases deben incluir, al menos, el punto inicial y cómo el software está siendo invocado. Esto permite que la cobertura de prueba sea verificada vía un rápido escaneo de los nombres de los métodos. También puede ser útil incluir los resultados esperados en el nombre del método de prueba, mientras esto no cause que el nombre sea demasiado largo para ver o leer.

También es buena idea poner a prueba tus pruebas. Puedes verificar que detectan el error al incluir dicho error en el código de producción (por supuesto, en una copia privada que desecharás). Asegúrate que reporte los errores de manera significativa. También debes verificar que tus pruebas hablan claramente a una persona que trata de entender tu código. La única manera de hacerlo es tener a alguien que no está

familiarizado con tu código para que lea tus pruebas y te diga qué ha aprendido. Escucha cuidadosamente lo que te diga. Si no entendió algo no es porque no sea muy brillante. Es más probable que tú no fueras muy claro. (¡Continúa e invierte los roles, lee sus pruebas!).

CONSEJO 96 DE 97

Preocúpate por el código

Por Pete Goodliffe · Traducción: Espartaco Palma

No hace falta ser Sherlock Holmes para saber que los buenos programadores escriben buen código. Los malos programadores… no. Ellos producen monstruosidades que el resto de nosotros tenemos que limpiar. ¿Tú quieres escribir las cosas buenas, verdad? Quieres ser un buen programador.

El buen código no rebota en el aire. No es algo que pasa por suerte cuando los planetas se alinean. El buen código tiene que ser trabajado, duramente. Y sólo obtendrás buen código si te preocupas por un buen código.

La buena programación no nace de la mera competencia técnica. He visto programadores altamente intelectuales que pueden producir intensos e impresionantes algoritmos, que conocen su lenguaje estándar de corazón, pero que escriben el código más horrible. Es doloroso de leer, doloroso de usar y doloroso de modificar. He visto programadores más humildes que se adhieren a un código muy sencillo, pero que escriben programas elegantes y expresivos, y es placentero trabajar con ellos.

Basado en mis años de experiencia en la fábrica de software, he concluido que la verdadera diferencia entre programadores adecuados y grandes programadores es esta: actitud. Los buenos programadores se dedican a tomar un enfoque profesional y quieren escribir el mejor software que puedan, aún con las limitaciones y presiones de la fábrica de software del mundo real.

El código del infierno está empedrado de buenas intenciones. Para ser un excelente programador tienes que estar más arriba de las buenas intenciones y realmente preocuparte por el código, fomentar perspectivas positivas y desarrollar actitudes sanas. El gran código es cuidadosamente confeccionado por maestros artesanos, no hackeado irreflexivamente por programadores flojos o erigido misteriosamente por autoproclamados gurús del código.

Tú quieres escribir buen código. Quieres ser un buen programador. Entonces, te preocupas por el código:

> En cualquier situación de codificación, te rehúsas a hackear algo que sólo parece que funciona. Te esfuerzas para elaborar un código elegante que es claramente correcto (y tienes buenas pruebas para mostrar que es correcto).
>
> Escribes código que es ***descubrible*** (que otros programadores pueden tomarlo y entenderlo fácilmente), que es mantenible (que tú u otros programadores serán capaz de modificarlo fácilmente en el futuro) y que es correcto (tomas todas las medidas posibles para determinar que has solucionado el problema, no sólo hacer que parezca que el programa funciona).
>
> Trabajas bien junto con otros programadores. Ningún programador es una isla. Pocos programadores trabajan solos; la mayoría trabaja en un equipo de programadores, ya sea en un entorno empresarial o en un proyecto de código abierto. Consideras a los otros programadores y construyes código que otros pueden leer. Deseas que el equipo escriba el mejor software posible, en lugar de hacerte lucir inteligente.

> Cada vez que tocas una pieza de código te esfuerzas en dejarlo mejor que como lo encontraste (ya sea mejor estructurado, mejor probado, más entendible).
>
> Te preocupas por el código y la programación, así que estás aprendiendo constantemente nuevos lenguajes, idiomas y técnicas. Pero sólo los aplicas cuando es apropiado.

Afortunadamente, estás leyendo esta colección de consejos porque te preocupas por el código, te interesa, es tu pasión. Te diviertes programando. Disfrutas cortar código para solucionar problemas difíciles. Produces software que te hace sentir orgulloso.

CONSEJO 97 DE 97

Tus clientes no quieren decir lo que dicen

Por Nate Jackson · Traducción: Espartaco Palma

Nunca he conocido a un cliente que no estuviera muy feliz de decirme qué es lo que quería, usualmente con gran detalle. El problema es que los clientes no siempre dicen toda la verdad. Generalmente no mienten, pero hablan en idioma cliente, no en idioma desarrollador. Usan sus términos y contextos. Dejan fuera detalles importantes. Suponen que has estado en su compañía por 20 años, igual que ellos. ¡Esto se agrava con el hecho de que muchos clientes realmente no saben lo que quieren en primer lugar! Algunos pueden tener un rasgo de la "visión global", pero rara vez son capaces de comunicar los detalles de sus visiones con efectividad. Otros podrían ser un poco claros en la visión completa, pero saben lo que no quieren. Entonces, ¿cómo es posible que puedas entregar un proyecto de software a alguien que no está diciendo toda la verdad acerca de lo que quiere? Es bastante simple. Sólo interactúa más.

Reta a tus clientes tempranamente y rétalos seguido. No te limites a repetir lo que dijeron que querían en sus palabras. Recuerda: ellos no quieren decir lo que te dijeron. Frecuentemente hago esto intercambiando palabras en conversaciones con ellos y juzgando sus reacciones. Estarás sorprendido de cuántas veces el término cliente tiene un significado completamente diferente al término comprador. Sin embargo, el hombre diciéndote qué quiere en su proyecto de software usará los términos indistintamente y espera que sigas el rastro de a cuál se refiere. Te confundirás y el software que escribas sufrirá.

Discute los temas numerosas veces con tus clientes antes de que decidas que has entendido lo que quieren. Intenta reformular el problema dos o tres veces con ellos. Háblales acerca de las cosas que suceden justo antes o justo después del tópico del que están hablando para obtener un mejor contexto. Si es posible, ten a varias personas hablándote del mismo tema en conversaciones separadas. Casi siempre te dirán historias diferentes, las cuales descubrirán hechos separados pero relacionados. Dos personas hablándote sobre el mismo tema se contradicen frecuentemente. Tu mayor oportunidad de éxito es discutir a fondo las diferencias antes de comenzar la elaboración de tu ultracomplejo software.

Haz uso de ayudas visuales en tus conversaciones. Esto podría ser tan sencillo como usar una pizarra en una reunión, tan fácil como crear un maqueta visual en la fase de diseño o tan complejo como elaborar un prototipo funcional. Es conocido que usar ayudas visuales durante una conversación ayuda a prolongar nuestro periodo de atención e incrementa la tasa de retención de la información. Toma ventaja de este hecho y configura tu proyecto para el éxito.

En una vida anterior era un “programador multimedia” en un equipo que producía proyectos ostentosos. Un cliente nuestro describió sus pensamientos con el look & feel del proyecto con gran detalle. El esquema general de colores discutido en las reuniones de diseño indicaba un fondo negro para la presentación. Pensábamos que lo teníamos hecho. Los equipos de diseñadores gráficos comenzaron a producir cientos de capas de archivos gráficos. Un montón de tiempo fue invertido moldeando el producto final. Una sorprendente revelación fue hecha el día en que mostramos al cliente el fruto de nuestra labor. Al ver el producto, las palabras exactas sobre el color de fondo fueron: “Cuando dije negro, me refería a blanco”. Así que, ya ves, nunca es tan claro como el blanco y negro.