**Enredados en herramientas**

**Qué hay de malo en las bibliotecas y qué hacer al respecto**

**por Mike Taylor**

# Científicos contra ingenieros

Respuestas a "¿Qué pasó?" Los artículos parecían dividirse aproximadamente entre 50 y 50 entre científicos e ingenieros. Los científicos dijeron: "Sí, sí, sé exactamente lo que quieres decir"; Los ingenieros dijeron: "Esta es la reutilización que todos hemos intentado lograr durante los últimos cuarenta años". Una parte de mí sabe que los ingenieros tienen razón, o al menos en parte tienen razón; pero una parte de mí siente que seguramente este no puede ser el brillante futuro digital que todos estábamos esperando. ¿La reutilización realmente tiene que ser tan, bueno, asquerosa ? Seguramente "Aprender estas 57 clases" no es la reutilización que nos prometieron.

Pero la verdad es que, como la mayoría de nosotros, llevo sombreros de científico e ingeniero en diferentes momentos. Lo que quiero con un sombrero puesto puede no ser lo que quiero con el otro sombrero puesto. Hay muchas cosas en las que ambos sombreros están de acuerdo: la conveniencia de la simplicidad, la claridad y la generalidad, por ejemplo, las tres palabras mágicas en la portada del libro de Kernighan y Pike La práctica de la programación.. Pero los sombreros tienen ideas diferentes sobre cómo llegar allí. El sombrero de la ciencia se preocupa profundamente por la comprensión. No, es más que eso: se preocupa por una comprensión profunda. El ingeniero se preocupa por hacer las cosas. Lo que todavía estoy tratando de resolver es esto: ¿es más rápido hacer las cosas sin molestarme en entender, o vale la pena invertir tiempo en comprenderlo incluso cuando se mide de una manera puramente utilitaria? Honestamente no lo se.

# Eso era entonces, esto es ahora

No solía ser así. He estado programando computadoras desde 1980: Commodore BASIC, luego C, luego Perl, y más recientemente Ruby. En el camino he usado muchos otros idiomas, pero esos han sido los cuatro que, en algún momento u otro, han sido mis favoritos, mi idioma de elección. Durante unos años atrás en la bruma del tiempo, todos mis pensamientos tenían números de línea. En la década de 1990, cada vez que quería escribir un programa, mis dedos escribían automáticamente int main (int argc, char \* argv []) tan pronto como mi cerebro posterior detectaba un nuevo buffer de emacs. Pasé una década de mi vida creyendo que la distinción entre contexto escalar y de lista tenía sentido, y que wantarray ()era el tipo de cosas que podría incluir un lenguaje racional. Ahora estoy mejor, pero a veces me parece extraño que los objetos físicos en el espacio de la carne no sepan responder a cada mensaje con un cierre adjunto.

En los últimos 30 años, he visto muchos cambios en la forma en que se realiza la programación, pero quizás la clave es el surgimiento de la ingeniería. En un Commodore 64 que tenía que ser científico para hacer nada. Tenías que conocer la máquina profundamente, tenías que asimilar su núcleo más íntimo, convertirte en uno con sus parámetros operativos, resonar con las frecuencias sutiles de sus profundidades más profundas o, al menos, saber en qué lugar PODER para girar la frontera. púrpura.

Lo mismo era cierto cuando se programaba en C en Unix: aún necesitabas conocer la máquina, incluso si ahora era una máquina un poco más abstracta. Pero hubo un cambio importante, en retrospectiva, incluso más importante que el cambio de código de espagueti no estructurado a estructurado en bloque, o de interpretado a compilado: había una biblioteca. Sí, una biblioteca, solo una. (Bueno, aproximadamente uno; estoy simplificando un poco, porque también hubo maldiciones , termcapy algunos otros pequeños). La biblioteca proporcionó formas simples de hacer cosas complejas, como ordenar y dar salida formateada, así como operaciones a nivel de sistema como configurar tuberías y procesos de bifurcación. No hace falta decir que todos estuvieron de acuerdo en que la biblioteca era una buena cosa, un gran ahorro de tiempo. Así que todos aprendimos la biblioteca: los buenos programadores de C sabían todo, incluyendo cosas como la diferencia en el orden de los argumentos entre write () y fwrite () , la firma de una función de comparación sort () y cuándo usar strcpy () vs. memcpy () . Dio una paleta completamente nueva para pintar.

Y luego, wow, Perl. Perl no ofrece una biblioteca. Perl tiene el que probablemente sea el catálogo más grande de bibliotecas de cualquier idioma del mundo: CPAN (la [red integral de archivos de Perl](http://search.cpan.org/) . En el momento de la redacción, CPAN ofrece 19941 distribuciones para descargar, de las cuales 500 pertenecen solo a XML. Es abrumador. En los últimos doce años, se han escrito al menos tres biografías con el título El último hombre que lo sabía todo (sobre Athanasius Kircher, Thomas Young y Joseph Leidy). Es emocionante pensar en un momento en que era posible que un solo individuo dominar todo el conocimiento acumulado de la humanidad; ahora no es posible dominar todas las bibliotecas de código abierto para Perl Hay tantas zurcidos. mucho de todo.

# Ceguera de la biblioteca

Entonces, ¿qué debe hacer un niño? Elija las bibliotecas con cuidado, aprenda solo algunas y espere que haya elegido las correctas. Por razones históricas, hago mi manejo de XML en Perl usando la torpemente llamada biblioteca XML :: LibXML , y no tengo una idea real de si es la mejor para el trabajo. Es el más fácil de usar para mí, porque todo lo que tengo que hacer es encontrar algún código donde lo usé antes, cortar y pegar la instanciación / invocación, y ajustar para que se ajuste al caso actual. Pero si bien mi gorra de ingeniero está bastante contenta con eso, mi gorra de científico está profundamente infeliz porque no entiendo lo que estoy haciendo. Realmente no.

Por supuesto, el problema de XML en Perl no sería tan difícil de resolver: solo tendría que tomarme un tiempo para navegar un poco, ver qué bibliotecas XML están usando otras personas y qué dicen sobre ellas, averiguar cómo El código está "vivo" (¿hay lanzamientos recientes y con qué frecuencia son?), Descárguelos e instálelos, pruébelos y vea cuál me gusta más. Si XML fuera el único área donde Library Blindness fuera un problema, probablemente simplemente me callaría y lo haría. Pero lo mismo se aplica en otras cien áreas, y estoy demasiado ocupado cortando árboles para detener y afilar mi hacha.

¿Estoy diciendo que no debería haber bibliotecas para Perl? Diablos, no. ¡No quiero hacer mi propio análisis XML! ¿Entonces que? ¿Que debería haber One True Library, como había para C en la década de 1980, y alguien de confianza debería decidir qué hay dentro y qué no? No, eso nunca podría funcionar. En verdad, no estoy seguro de lo que soydiciendo: No sé la solución. Ni siquiera puedo proponer uno, más allá de la observación obvia de que mejores mecanismos de gestión de la reputación harían menos desalentador elegir una biblioteca de entre muchas que compiten en el mismo espacio. Necesitamos bibliotecas, y muchas de ellas, por la sencilla razón de que se espera que hagamos mucho más de lo que estábamos cuando teníamos Commodore 64 (o, Dios nos ayude, VIC-20). Los pequeños programas autónomos que escribimos no necesitaban bibliotecas porque no hacían llamadas REST a servicios web, ni asignaban objetos a bases de datos relacionales, ni transformaban documentos XML. (Quizás podríamos haber usado bibliotecas de gráficos; pero dado que el hardware realmente no admitía el trazado arbitrario, la mayoría de los gráficos se hicieron con caracteres definidos por el usuario y, si tuvo suerte, sprites).

Entonces las bibliotecas son necesarias. ¿Son un mal necesario? Solo si crees que la maldad es una necesidad. No creo que lo sea; Pero la evidencia experimental está en mi contra.

# La mentira de la biblioteca

Un comentarista llamado Silversmith en la discusión de Reddit de mi artículo original tenía una visión idealizada de las bibliotecas: “Tienes el problema X, que consiste en los subproblemas X1, X2, X3. Hay soluciones disponibles para X1 y X3. Si no tiene ganas de ser albañil, codifique X. Elijo codificar X2, enchufar las soluciones X1 y X3, y pasar el resto del día investigando y posiblemente resolviendo el problema Y ". Eso es lo que yo llamo la mentira de la biblioteca, y creo que Silversmith se la ha tragado. La suposición tácita es que "conectar las soluciones X1 y X3" es trivial, que requiere poco esfuerzo y casi nada de tiempo, y que el resultado es una X agradable, limpia e integrada. Pero sabemos por experiencia que esto no es cierto. El XML que produce X1 supuestamente tiene el mismo formato que X3 requiere, pero es misteriosamente rechazado cuando lo alimenta a X3 (y, por supuesto, no hay ningún mensaje de error útil, solo "error XML"). Proporciona dos funciones de enlace para que X3 devuelva la llamada, pero una nunca recibe una llamada y la otra parece llamarse dos veces ... a veces, en condiciones que no están claras. Y así sigue, y entonces te encuentras escribiendo no solo X2 sino también capas envolventes X1 'y X3' que se supone que hacen que X1 y X3 se vean como quieres. E incluso cuando haya terminado, no tiene muy claro por qué el agregado funciona para los casos que ha probado; y no tiene confianza en que algo no saldrá misteriosamente mal cuando comience a usarlo en otros casos. a veces, en condiciones que no están claras. Y así sigue, y entonces te encuentras escribiendo no solo X2 sino también capas envolventes X1 'y X3' que se supone que hacen que X1 y X3 se vean como quieres. E incluso cuando haya terminado, no tiene muy claro por qué el agregado funciona para los casos que ha probado; y no tiene confianza en que algo no saldrá misteriosamente mal cuando comience a usarlo en otros casos. a veces, en condiciones que no están claras. Y así sigue, y entonces te encuentras escribiendo no solo X2 sino también capas envolventes X1 'y X3' que se supone que hacen que X1 y X3 se vean como quieres. E incluso cuando haya terminado, no tiene muy claro por qué el agregado funciona para los casos que ha probado; y no tiene confianza en que algo no saldrá misteriosamente mal cuando comience a usarlo en otros casos.

No digo que construir X1 'y X3' y lidiar con todo el dolor no sea aún más rápido en general que escribir tus propios X1 y X3. Pero digo que necesitamos al menos ser honestos con nosotros mismos acerca de cuánto tiempo tomará “simplemente conectar las soluciones X1 y X3”; y debemos reconocer que el resultado final podría no ser tan manejable o confiable como si hubiéramos creado la solución completa.

Las bibliotecas son una victoria. Pero no son una victoria tan grande como quieren que pienses, y a veces son el tipo de victoria que te hace desear haber perdido.

# Veré la ceguera de tu biblioteca y te elevaré la fiebre del marco

Si hay algo más aterrador que mapear a través de un laberinto de pequeñas bibliotecas retorcidas, todo diferente, se está atascando en el pantano de framework. "Marco" es una palabra que ha saltado a la fama en los últimos años: rara vez la escuchaste antes del cambio de milenio, al menos fuera de los círculos de Java, pero ahora son el mejor juego para jugar. A veces la palabra parece ser simplemente un sinónimo de moda para biblioteca, pero el marco adecuado es una bestia más grande y más peluda.

Un marco se puede definir como un conjunto de bibliotecas que dicen "No nos llames, te llamaremos". Cuando invocas una biblioteca tradicional, aún tienes el control: haces las llamadas a la biblioteca que deseas hacer y lidias con las consecuencias. Un marco invierte el flujo de control: se lo entrega y espera a que invoque las diversas funciones de devolución de llamada que proporciona. Pones la vida de tu programa en sus manos. Eso tiene consecuencias: una de las más importantes es que, si bien su programa puede usar tantas bibliotecas como quiera, solo puede usar, o más bien, puede ser usado por un marco. Los marcos son celosos. No comparten

¿Vale la pena ceder el control a un marco? Bueno, a veces. Mi experiencia ha sido que mientras hagas las cosas a la manera del marco, y no necesites ir más allá de lo que el autor imaginó, lo harás bien: las cosas funcionarán rápida y simplemente y el sol brillará y los pájaros cantarán hasta De repente, ¡uh-oh! ¡Necesitas hacer algo un poco diferente y wham! Todo se desmorona. Dependencias sutiles que te ocultaban cuando todo iba bien saltan repentinamente de los arbustos y te gritan. Los monstruos que estaban encadenados en el sótano se sueltan todos a la vez, y de repente agregas esta línea a tu devolución de llamada y eso hace quealgo aparentemente no relacionado sale mal. Y entonces parches para eso y algo completamente separado sale mal en el otro extremo de su aplicación, y el archivo de registro no le dice nada útil. Entonces intenta buscar el área problemática en la documentación, pero cuando encuentra la entrada para la función newRecordAddedHook , simplemente dice "Este gancho se llama cuando se agrega un nuevo registro".

Hay un ejemplo aterrador de fallas en el marco en el libro de Rails, [Desarrollo web ágil con Rails (tercera edición)](http://pragprog.com/titles/rails3/agile-web-development-with-rails-third-edition) . El tutorial ha estado funcionando durante 187 páginas, enseñándole cómo hacer cosas como agregar has\_many: line\_items a la clase de modelo de Producto: código mínimo, todo se explica por sí mismo. Entonces, de repente, de la nada, el capítulo de Internacionalización le dice que simplemente agregue un formulario al diseño de la tienda:

|  |  |
| --- | --- |
|  | <% form\_tag *''*, method => *'GET'*, :class => *'locale'* do &> |
|  | <%= select\_tag *'locale'*, options\_for\_select(LANGUAGES, I18n\_locale), |
|  | :onchange => *'this.form.submit()'* %> |
|  | <%= submit\_tag *'submit'* %> |
|  | <%= javascript\_tag “$$(*'.locale input'*).each(Element.hide)” %> |

Quiero decir, ¿qué?

La etiqueta de formulario, la etiqueta de selección y la etiqueta de envío tienen sentido a la luz de lo que había sucedido antes, pero ese trozo de JavaScript aparece en el texto completamente formado, directamente desde la mente de Zeus. Es genial que aparezca allí mismo en el tutorial, pero no hay forma de que haya llegado a eso por mí mismo.

Cuando me encuentro con este tipo de cosas, siento que estoy jugando nuevamente a los juegos de aventura de los 80. Recuerda: ABRIR PUERTA / La puerta está cerrada / DESBLOQUEAR LA PUERTA / ¿Abrir la puerta con qué? / DESBLOQUEAR PUERTA CON LLAVE / ¿Te refieres a la llave de hierro o la llave de latón? / DESBLOQUEAR LA PUERTA CON LA LLAVE DE LATÓN / La llave de latón no encaja. Son conjeturas. Peor que eso, es adivinar detalles poco interesantes como el vocabulario en lugar de adivinar sobre el problema real. Cuando jugué Lurking HorrorTuve problemas terribles con un rompecabezas cerca del final, cuando era obvio que algo estaba en un charco de agua y tenía que sacarlo. No pude ENTRAR A LA PISCINA o NADAR EN LA PISCINA, y los intentos de PONER LAS MANOS EN LA PISCINA o SENTIR LA PISCINA o SENTIRSE EN LA PISCINA fueron infructuosos. Al final me topé con la respuesta, LLEGAR A LA PISCINA. Este tipo de juego de adivina qué tenía el autor en mente es lo suficientemente desalentador cuando intentas derrotar un horror gótico desde fuera del tiempo y salvar al mundo del dominio malvado de las bestias de culto; pero es simplemente tonto cuando todo lo que intentas hacer es mostrar un menú desplegable.

# El problema no son las bibliotecas. El problema son las bibliotecas malas

He pintado un cuadro negro. Reconozco la necesidad en estos días de escribir programas que hagan mucho más de lo que hicieron en la década de 1980; He admitido que no podemos hacer eso sin bibliotecas; Es posible que no haya salido y dije que los marcos también son necesarios, pero en el fondo de mi corazón sé que es verdad (de lo contrario, ¿por qué me molestaría en leer el libro de Rails?) Es gracioso recordar a mis profesores, cuando estaba haciendo mi licenciatura en matemáticas y CS a fines de la década de 1980, hablando de la "crisis del software", realmente no tenían idea de lo malo que iba a ser. Las palabras de Dijkstra del programador humildeparece profético: "mientras no hubiera máquinas, la programación no era un problema en absoluto [...] ahora tenemos computadoras gigantes, la programación se ha convertido en un problema igualmente gigantesco". Es difícil de creer que fue escrito en 1972.

Así que parece que me estoy quejando bastante. ¿Tengo alguna sugerencia constructiva?

¡Porque?, si! Sí. Tengo un manifiesto de dos puntas.

# Rama 1: sencillo, más claro, más corto Documentación

Lo más simple que podemos hacer en este momento es repensar cómo documentamos las bibliotecas. Las herramientas de documentación automática como Javadoc y Rdoc son buenas para producir gruesas pilas de papel, pero no tan buenas para decirles a los usuarios de la biblioteca lo que necesitan saber. Son la fuente de la mayoría de la documentación " newRecordAddedHook del mundo cuando se agrega un nuevo registro". Pueden ser útiles, pero no son un sustituto para escribir sobre la biblioteca: resúmenes de una página que responden a las tres preguntas clave: qué hace la biblioteca, por qué debe usarla y cómo hacerlo. Para los puntos de bonificación, el resumen de una página debe evitar el uso de la palabra "empresa" (a menos que sea una biblioteca de Star Trek buques) e "innovador".

Aquí hay un ejemplo del tipo de cosas de las que debemos alejarnos: cito textualmente del primer párrafo de contenido en la página [principal del sitio web principal sobre JavaServer Faces](http://java.sun.com/javaee/javaserverfaces/) :

*Desarrollado a través del Proceso de la Comunidad Java bajo JSR - 314, la tecnología JavaServer Faces establece el estándar para construir interfaces de usuario del lado del servidor. Con las contribuciones del grupo de expertos, las API de JavaServer Faces se están diseñando para que puedan ser aprovechadas por herramientas que facilitarán aún más el desarrollo de aplicaciones web. Varios proveedores de herramientas respetados eran miembros del grupo de expertos JSR-314, que desarrolló la especificación JavaServer Faces 1.0. Estos proveedores se comprometen a admitir la tecnología JavaServer Faces en sus herramientas, promoviendo así la adopción del estándar de tecnología JavaServer Faces.*

Este es un modelo de ofuscación; una cosa de belleza, a su manera. En casi cien palabras, no dice casi nada. Se las arregla para usar el nombre "JavaServer Faces" no menos de cinco veces, sin dar una sola pista más de lo que realmente es, algo que tiene que ver con las interfaces de usuario, aparentemente, ya sea en la Web o en el escritorio o en cualquier otro lugar. No pude decir. Se le dice un montón de cosas que puede que simplemente se han interesado en después de un año o dos de realidad trabajar con una tecnología, sino que ningún ser racional acercarse a ella por primera vez, posiblemente, podría preocuparse. Que varios vendedores de herramientas respetados fueran miembros del grupo de expertos JSR-314 no era el primero de mi lista de Cosas para averiguar sobre JSF.

(Permítanme decir que mi punto aquí no es molestar a JSF en particular. Es simplemente uno de los cientos de ejemplos igualmente densos que podría haber elegido, y ese, realmente, es precisamente mi punto).

Sugiero que cualquier biblioteca cuyas características clave no se puedan resumir en una página de A4 (o Carta de EE. UU., Si insiste) es demasiado compleja y necesita ser rediseñada. Por "demasiado complejo" aquí, no me refiero simplemente a que hace demasiado, sino que lo que hace no está lo suficientemente enfocado. De hecho, deberías poder resumir brevemente lo que hace una biblioteca en una sola oración. Y si la oración comienza "Es un marco para construir abstracciones que pueden usarse para derivar modelos que ...", entonces pierdes.

# Punta 2: minimiza el radio de comprensión

"Radio de comprensión" es un nuevo término que estoy presentando aquí, porque describe un concepto importante para el que no creo que haya un nombre. Es una propiedad de una base de código definida de la siguiente manera: si está mirando un fragmento de código dado, qué tan lejos de ese fragmento de código debe tener en mente en ese momento para comprender el fragmento en cuestión ? Es una especie de medida de cuán buena es la encapsulación en toda la base de código, aunque cuando digo "encapsulación" aquí, estoy usando ese término en un sentido amplio que significa más que solo problemas técnicos, como qué proporción de miembros de datos son Marcado privado. Estoy hablando de un problema humano aquí (y, por lo tanto, lamentablemente, uno casi imposible de medir, aunque lo sabemos cuando lo vemos).

Entonces, por ejemplo: cuando estoy leyendo una base de código con la que aún no estoy familiarizado, si me encuentro con un objeto cuya clase se llama Employee o HashTable o CharacterEncoding , puedo estar razonablemente seguro de entender lo que esa clase debe ser haciendo, y puedo adivinar el significado de los métodos invocados en él: el radio de comprensión es agradablemente bajo. Si me encuentro con objetos de la clase EmployeeFactory , probablemente pueda adivinar y es probable que tenga razón. Si encuentro un CharacterEncodingMediatorFrobnicationDecorator, de ninguna manera voy a tener una idea clara de qué se trata a menos que vaya y lea el código por mí mismo, por lo que el radio de comprensión aumenta. El nombramiento de clases es solo uno de los muchos factores que contribuyen al radio de comprensión: otros incluyen la inmutabilidad de los objetos, funciones que están garantizadas sin efectos secundarios y, la más difícil de cuantificar, la unidad conceptual de los diversos módulos.

El significado del radio es obvio: cuando es bajo, necesitamos cargar solo un poco del código en nuestras mentes a la vez para comprender la parte que estamos tratando de leer, depurar o mejorar. Cuando el radio es alto, se debe dedicar un valioso espacio mental a asuntos que no son directamente relevantes para el problema que estamos tratando de resolver; y cuanto más tengamos que pensar al mismo tiempo, menos atención podremos prestarle al tema central.

El radio de comprensión tiene que ver con cuánto tienes que mantener en tu cabeza antes de que puedas comenzar a ser productivo. Joel Spolsky [ha escrito](http://www.joelonsoftware.com/articles/fog0000000043.html) sobre el problema de las interrupciones: cómo sacan a los programadores de "la zona" para que tengamos que volver a cargar todo nuestro estado mental antes de que podamos volver al trabajo útil (ver punto 8, "¿Los programadores tienen un trabajo tranquilo? condiciones? ”) Siendo esto así, necesitamos construir nuestro software de tal manera que los programadores tengan una cantidad mínima para recargar después de cada interrupción. Y esto es particularmente importante para los diseñadores de bibliotecas y (cuando sea posible) marcos.

¿Cómo podemos reducir el radio? En términos generales, es solo la navaja de Occam: no multiplique entidades sin necesidad. Cada biblioteca debe proporcionar el menor número posible de clases de API, y cada una debe admitir el número mínimo de métodos. Esto es obvio. ¿Pero podemos ir más allá de estas generalidades para proponer algunas pautas tentativas para reducir el radio? Podríamos intentar lo siguiente, con la humildad adecuada, con la esperanza de que su crítica conduzca a mejores reglas:

* Cada tipo de cosa debe tener una sola representación. Esto parece tan obvio que apenas es necesario indicarlo, pero la distinción errónea de Java entre Array y ArrayList hace que valga la pena declararlo.
* En la medida de lo posible, los objetos deben representar cosas reales. A veces, los patrones de diseño introducen clases para representar abstracciones como iteraciones, estrategias y decoradores: eso puede ser un mal necesario, pero de todos modos es un mal. Cada una de esas clases aumenta el radio.
* Los objetos inmutables son más fáciles de pensar que los mutables. Si se garantiza que un objeto es inmutable, entonces la probabilidad de que necesite leer su código fuente para comprenderlo se reduce considerablemente.
* Del mismo modo, las funciones sin efectos secundarios son más fáciles de pensar que las que cambian el estado de uno o más objetos.

Cuando los lenguajes tienen una sintaxis que nos permite marcar objetos como inmutables o funciones libres de efectos secundarios, debemos usarlos: dicha notación es documentación que el compilador puede verificar por nosotros. La ausencia de tales instalaciones de lenguajes extremadamente dinámicos como Ruby es una debilidad.

Finalmente, como Bjarne Stroustrup señaló en El lenguaje de programación C ++ (2ª edición) , “El diseño y la programación son actividades humanas; olvídalo y todo está perdido ... No hay métodos de 'libro de cocina' que puedan reemplazar la inteligencia, la experiencia y el buen gusto ". Creo que el radio de comprensión es un concepto útil, pero que nos servirá mejor si solo tenemos en cuenta al diseñar API, en lugar de seguir ciegamente las reglas que tienen la intención de reducirlo.

Al principio de mi carrera, pasé diez años trabajando en una base de datos patentada de texto y objetos, escrita completamente en C, que tenía que construir su orientación a objetos a partir de piezas de repuesto. Según todas las expectativas estándar, debería haber sido una pesadilla. Pero, de hecho, el código tenía un radio de comprensión muy bajo, simplemente porque fue diseñado con cuidado, atención y gusto por un arquitecto muy talentoso y diligente. Para trabajar en ese sistema tan grande, tenía que tener solo una parte muy pequeña en mi mente a la vez. Solo puedo esperar que dentro de diez años, las personas que han trabajado conmigo puedan decir lo mismo sobre mi código.