EDICIÓN ESPECIAL

El programador pragmático

Viaje a la maestría

DAVID THOMAS

ANDREW HUNT

Prologado por SARON YITBAREK

ANAYA MULTIMEDIA

Índice de contenidos

2 Un enfoque pragmático

Existen algunos consejos y trucos que se aplican a todos los niveles del desarrollo de software, procesos que son prácticamente universales e ideas que son casi axiomáticas. Sin embargo, estos enfoques rara vez se documentan como tales; lo más habitual es encontrarlos escritos como frases sueltas en discusiones sobre diseño, gestión de proyectos o creación de código. Pero, para su conveniencia, hemos reunido aquí estas ideas y procesos.

El primer tema, y quizá el más importante, llega al corazón del desarrollo de software: "La esencia del buen diseño". Todo lo demás sigue a partir de ahí.

Las dos secciones siguientes, "DRY: los males de la duplicación" y "Ortogonalidad", están bastante relacionados. El primero le advierte que no duplique la información a lo largo del sistema y el segundo que no divida una sola porción de información entre múltiples componentes del sistema.

A medida que aumenta el ritmo del cambio, se va haciendo cada vez más difícil conseguir que nuestras aplicaciones sigan siendo relevantes. En "Reversibilidad", echaremos un vistazo a algunas técnicas que ayudan a aislar nuestros proyectos de su entorno cambiante.

Las siguientes dos secciones también están relacionadas. En "Balas trazadoras", hablamos de un estilo de desarrollo que permite reunir requisitos, probar diseños e implementar código al mismo tiempo. Es la única forma de seguir el ritmo de la vida moderna.

Seguro frente a la obsolescencia

Las formas de datos legibles para los humanos y los datos autodescriptivos sobrevivirán a todas las demás formas de datos y las aplicaciones que las crearon. Punto. Mientras los datos sobrevivan, tendrá la oportunidad de usarlos; posiblemente, mucho tiempo después de que la aplicación original con la que se escribieron haya desaparecido. Puede analizar sintácticamente un archivo así con solo un conocimiento parcial de su formato; con la mayoría de archivos binarios, debe conocer todos los detalles del formato entero para poder analizarlo sintácticamente con éxito.

Piense en un archivo de datos de algún sistema heredado que se le da.[1] Sabe muy poco de la aplicación original; lo único que es importante para usted es que mantenía una lista de números de la Seguridad Social de los clientes, que tiene que encontrar y extraer. Entre los datos, ve:

```
<FIELD10>123-45-6789</FIELD10>
...
<FIELD10>567-89-0123</FIELD10>
...
<FIELD10>901-23-4567</FIELD10>
```

Al reconocer el formato de un número de la Seguridad Social, puede escribir enseguida un programa pequeño para extraer esos datos, incluso si no tiene información sobre nada más en el archivo.

Pero imagine que el archivo hubiese tenido este formato en vez del anterior:

```
AC27123456789B11P
...
XY43567890123QTYL
...
6T2190123456788AM
```

Puede que no hubiese reconocido el significado de los números con tanta facilidad. Esa es la diferencia entre datos legibles para los humanos y datos comprensibles para los humanos. Y ya que estamos, `FIELD10` tampoco ayuda mucho. Algo como:

```
<SOCIAL-SECURITY-NO>123-45-6789</SOCIAL-SECURITY-NO>
```

hacen que el ejercicio sea obvio y garantiza que los datos sobrevivirán a cualquier proyecto que los crease.

Aprovechamiento

Casi todas las herramientas del universo de la informática, desde los sistemas de control de versiones a los editores y las herramientas de línea de comandos, pueden manejar texto simple.

1. Todo software se convierte en software heredado en cuanto se escribe.

Por ejemplo, supongamos que tiene un despliegue a producción de una aplicación grande con un archivo de configuración complejo específico para un sitio. Si el archivo está en texto simple, podría colocarlo bajo un sistema de control de versiones (consulte el tema 19 "Control de versiones"), de manera que mantenga un historial de todos los cambios de forma automática. Las herramientas de comparación de archivos como `diff` y `fc` le permiten descubrir de un solo vistazo qué cambios se han realizado, mientras que `sum` le permite generar un *checksum* para monitorizar el archivo en busca de modificaciones accidentales (o maliciosas).

La filosofía de Unix

Unix es famoso por estar diseñado en torno a la filosofía de herramientas pequeñas y afiladas, cada una de ellas pensada para hacer una cosa bien. Esta filosofía se pone en marcha mediante el uso de un formato subyacente común, el archivo de texto simple orientado a líneas. Las bases de datos usadas para la administración de sistemas (usuarios y contraseñas, configuración de red, etc.) se mantienen en archivos de texto simple. (Algunos sistemas también mantienen una forma binaria de determinadas bases de datos como optimización del rendimiento. La versión en texto simple se mantiene como una interfaz de la versión binaria).

Cuando un sistema falla, puede que nos encontremos solo con un entorno mínimo para restaurarlo (puede que no podamos acceder a los controladores de gráficos, por ejemplo). Situaciones como esta pueden hacer que aprecie más la sencillez del texto simple. Además, es más fácil realizar búsquedas en archivos de texto simple. Si no recuerda qué archivo de configuración gestiona las copias de seguridad de su sistema, el uso rápido de `grep -r backup /etc` debería indicárselo.

Realización de pruebas más fácil

Si utiliza texto simple para crear datos sintéticos para controlar las pruebas del sistema, solo es cuestión de añadir, actualizar o modificar los datos de prueba sin tener que crear ninguna herramienta especial para hacerlo. De manera similar, la salida en texto simple de pruebas de regresión puede analizarse de manera trivial con comandos del *shell* o un simple *script*.

Mínimo común denominador

Incluso en el futuro de agentes inteligentes basados en *blockchain* que recorren la salvaje y peligrosa Internet de forma autónoma, negociando el intercambio de datos entre ellos, el omnipresente archivo de texto seguirá estando ahí. De hecho, en entornos heterogéneos, las ventajas del texto simple pueden pesar más que los inconvenientes. Tiene que garantizar que todas las partes pueden comunicarse utilizando un estándar común. El texto simple es ese estándar.

Y, por supuesto, una buena manera de mantener la flexibilidad es escribir menos código. Cambiar el código abre la posibilidad de que se introduzcan fallos nuevos. "Configuración" explicará cómo extraer detalles del código por completo, donde puedan modificarse de manera más segura y sencilla.

Todas estas técnicas le ayudarán a escribir código que se adapte y no se rompa.

28 Desacoplamiento

Cuando tratamos de elegir algo por sí mismo, lo encontramos atado a todo lo demás en el universo.

—John Muir, Mi primer verano en la Sierra.

En el tema 8, "La esencia del buen diseño", afirmamos que utilizar buenos principios de diseño hará que el código que escribimos sea fácil de cambiar. El acoplamiento es el enemigo del cambio, porque vincula cosas que deben cambiar en paralelo. Esto hace que el cambio sea más difícil: dedicamos tiempo a rastrear todas las partes que necesitan cambiarse o lo dedicamos a preguntarnos por qué las cosas se estropearon cuando modificamos "solo una cosa" y no las demás con las que estaba acoplada.

Cuando diseñamos algo que queremos que sea rígido, como un puente o una torre, quizá, acoplamos los componentes (figura 5.1):

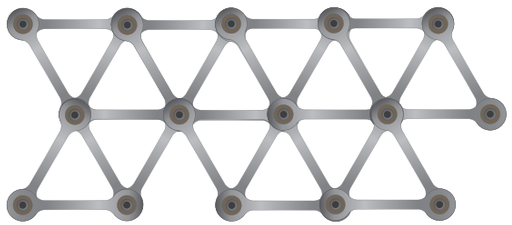

Figura 5.1.

Las piezas trabajan juntas para hacer que la estructura sea rígida. Compare eso con algo como esto (figura 5.2):

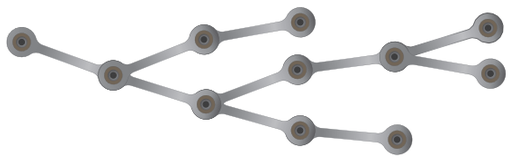

Figura 5.2.

Aquí no hay rigidez estructural: las piezas individuales pueden cambiar y otras solo se acomodan.

Si diseñamos puentes, queremos que mantengan su forma; necesitamos que sean rígidos. Pero, si diseñamos software que querremos modificar, queremos justo lo contrario: queremos que sea flexible. Y, para que sea flexible, los componentes individuales deberían acoplarse con la menor cantidad posible de otros componentes.

Para empeorar las cosas, el acoplamiento es transitivo: si A está acoplado con B y C, y B está acoplado con M y N, y C con X e Y, entonces en realidad A está acoplado con B, C, M, N, X e Y. Eso significa que hay un principio simple que debería seguir:

Truco 44. El código desacoplado es más fácil de cambiar.

Puesto que por lo general no escribimos código utilizando vigas de acero y remaches, ¿qué significa desacoplar código? En esta sección hablaremos de:

- ***Train wrecks* (choque de trenes):** Cadenas de llamadas a métodos.
- **Globalización:** Los peligros de las cosas estáticas.
- **Herencia:** Por qué la creación subclases es peligrosa.

Hasta cierto punto, esta lista es artificial: el acoplamiento puede producirse en cualquier momento en que dos porciones de código compartan algo, así que, mientras lea lo que se explica a continuación, preste atención a los patrones subyacentes para poder aplicarlos a su código. Y manténgase atento a cualquier señal de acoplamiento:

- Dependencias absurdas entre módulos o bibliotecas no relacionados.
- Cambios "simples" en un módulo que se propagan a través de módulos no relacionados en el sistema o estropean algo en cualquier otra parte de este.
- Desarrolladores que tienen miedo de modificar el código porque no están seguros de qué puede verse afectado.
- Reuniones a las que tiene que asistir todo el mundo porque nadie está seguro de a quién afectará el cambio.

Train Wrecks

Todos hemos visto (y probablemente escrito) código como este:

```
public void applyDiscount(customer, order_id, discount) {
  totals = customer
           .orders
           .find(order_id)
```

No escriba código dodo

Sin configuración externa, el código no es tan adaptable ni flexible como debería. ¿Es eso malo? Bueno, en el mundo real, las especies que no se adaptan mueren.

El dodo no se adaptó a la presencia de los humanos y su ganado en la isla Mauricio y se extinguió enseguida.[9] Se trata de la primera extinción documentada de una especie por la mano del hombre. No deje que su proyecto (o su carrera) acabe como el dodo.

Figura 5.11.

Las secciones relacionadas incluyen

- Tema 9, "DRY: los males de la duplicación".
- Tema 14, "Lenguajes de dominio".
- Tema 16, "El poder del texto simple".
- Tema 28, "Desacoplamiento".

No se exceda

En la primera edición de este libro, sugeríamos utilizar configuración en lugar de código de una manera similar, pero parece que deberíamos haber sido un poco más específicos en nuestras explicaciones. Cualquier consejo puede llevarse al extremo o usarse de forma inadecuada, así que veamos algunas advertencias:

No se exceda. Uno de nuestros primeros clientes decidió que todos y cada uno de los campos de su aplicación debería ser configurable. Como resultado, se necesitaban semanas para hacer incluso los cambios más pequeños, porque había que implementar tanto el campo como el código de administrador para guardarlo y editarlo. Tenían unas 40.000 variables de configuración y una pesadilla de código entre manos.

No pase decisiones a configuración por pereza. Si hay un debate genuino acerca de si una característica debería funcionar de esta manera o de aquella, o si debería ser elección de los usuarios, pruébela de una manera y reciba *feedback* acerca de si la decisión ha sido buena.

9. No ayudó que los colonos golpeasen a estos tranquilos (es decir: estúpidos) pájaros por diversión.

En medio de toda esta teoría, no olvide que hay también consideraciones prácticas. Puede que el tiempo de ejecución parezca aumentar de forma lineal para conjuntos de entrada pequeños, pero, si introduce en el código millones de registros, de pronto el tiempo se degradará y el sistema empieza a fallar. Si prueba una rutina de ordenamiento con claves de entrada aleatorias, puede que se sorprenda la primera vez que encuentre una entrada ordenada. Intente cubrir tanto los aspectos teóricos como los prácticos. Después de todas estas estimaciones, el único tiempo que importa es la velocidad del código, ejecutándose en el entorno de producción, con datos reales. Esto nos lleva a nuestro siguiente truco.

Truco 64. Pruebe sus estimaciones.

Si es difícil conseguir tiempos precisos, utilice perfiladores de código para contar el número de veces que se ejecutan los diferentes pasos de su algoritmo y trace estas cifras en relación con el tamaño de la entrada.

El mejor no es siempre lo mejor

También necesita ser pragmático respecto a la elección de los algoritmos apropiados; el más rápido no siempre es el mejor para la tarea. Si se le da un conjunto de entrada pequeño, un ordenamiento por inserción directo tendrá un rendimiento tan bueno como el del ordenamiento rápido y le llevará menos tiempo escribirlo y depurarlo. Además, debe tener cuidado si el algoritmo que elige tiene un coste de configuración elevado. Para conjuntos de entrada pequeños, esta configuración puede empequeñecer el tiempo de ejecución y hacer que el algoritmo no sea apropiado.

Preste atención también a la optimización prematura. Siempre es buena idea asegurarse de que un algoritmo es de verdad un cuello de botella antes de dedicar un tiempo valioso a intentar mejorarlo.

Las secciones relacionadas incluyen

- Tema 15, "Estimaciones".

Retos

- Todo desarrollador debería tener nociones acerca de cómo se diseñan y analizan los algoritmos. Robert Sedgewick ha escrito varios libros accesibles sobre la materia (*Algorithms* [SW11], *An Introduction to the Analysis of Algorithms* [SF13] y otros). Le recomendamos que añada uno de sus libros a su colección y ponga empeño en leerlo.

- Aquellos que quieran algo más detallado que lo que ofrece Sedgewick pueden leer los libros definitivos de Donald Knuth *El arte de programar ordenadores*, que analizan una amplia variedad de algoritmos.
- En el primer ejercicio que sigue nos fijamos en matrices de ordenamiento de enteros largos. ¿Cuál es el impacto si las claves son más complejas y la tara de la comparación de claves es alta? ¿Afecta la estructura de las claves a la eficiencia de los algoritmos de ordenamiento o el ordenamiento más rápido siempre es el más rápido?

Ejercicios

Ejercicio 28

Hemos creado código para un conjunto de rutinas de ordenamiento simples[5] en Rust. Ejecútelas en varios ordenadores a su disposición. ¿Siguen sus cifras las curvas esperadas? ¿Qué puede deducir acerca de las velocidades relativas de sus ordenadores? ¿Cuáles son los efectos de las distintas configuraciones para la optimización del compilador?

Ejercicio 29

En "Estimación con sentido común" decíamos que una búsqueda binaria es $O(\lg n)$. ¿Puede demostrarlo?

Ejercicio 30

En la figura 7.1 afirmábamos que $O(\lg n)$ es lo mismo que $O(\log_{10} n)$ (o, en realidad, logaritmos de cualquier base). ¿Puede explicar por qué?

40 Refactorización

Cambio y decadencia en todo lo que veo...

—H. F. Lyte, Abide With Me

A medida que evoluciona un programa, va haciéndose necesario replantearse decisiones anteriores y reelaborar porciones del código. Este proceso es perfectamente natural. El código necesita evolucionar; no es una cosa estática.

5. `https://media-origin.pragprog.com/titles/tpp20/code/algorithm_speed/sort/src/main.rs`.

8 Antes del proyecto

Al principio de un proyecto, usted y su equipo necesitan conocer los requisitos. Que les digan solo qué hacer o escuchar a los usuarios sin más no es suficiente: lea "El pozo de los requisitos" y aprenda cómo evitar las trampas y los obstáculos habituales.

La sabiduría popular y la gestión de restricciones son los temas de "Resolver rompecabezas imposibles". No importa si está ocupándose de los requisitos, el análisis, la escritura del código o las pruebas, aparecerán problemas difíciles de la nada. La mayor parte del tiempo, no serán tan difíciles como parecen.

Y, cuando surge ese proyecto imposible, nos gusta sacar nuestra arma secreta: "Trabajar juntos". Y con "trabajar juntos" no nos referimos a compartir documentos de requisitos masivos, enviar correos electrónicos con un montón de gente en copia o soportar reuniones eternas. Nos referimos a resolver problemas juntos mientras escribimos código. Le mostraremos a quién necesita y cómo empezar.

Aunque el Manifiesto Ágil empieza con "Individuos e interacciones sobre procesos y herramientas", prácticamente todos los proyectos "ágiles" comienzan con una discusión irónica acerca de qué procesos y qué herramientas se van a utilizar. Pero no importa lo bien planeado que esté, y al margen de qué "buenas prácticas" incluya, ningún proyecto puede sustituir al pensamiento. No necesita ningún proyecto o herramienta en particular, lo que necesita es "La esencia de la agilidad".

Con estas cuestiones cruciales resueltas antes de que el proyecto se ponga en marcha, puede estar mejor posicionado para evitar la "parálisis del análisis" y comenzar (y completar) de verdad su proyecto con éxito.

La talla única no le queda bien a nadie

La finalidad de una metodología de desarrollo de software es ayudar a las personas a trabajar juntas. Como explicamos en "La esencia de la agilidad", no hay un solo plan que pueda seguirse cuando se desarrolla software, sobre todo, si es un plan que se le ha ocurrido a alguien en otra empresa.

Muchos programas de certificación son aún peor que eso: se basan en que el alumno sea capaz de memorizar y seguir las reglas. Pero eso no es lo que le conviene. Necesita la capacidad de ver más allá de las reglas existentes y explotar las posibilidades para conseguir ventaja. Es una mentalidad muy diferente a "pero Scrum/Lean/Kanban/XP/el desarrollo ágil lo hace así...", etc.

En vez de eso, le conviene tomar los mejores elementos de una metodología en particular y adaptarlos para su uso. No hay una talla única, y los métodos actuales están muy lejos de ser completos, así que tendrá que fijarse en más de un método popular.

Por ejemplo, Scrum define algunas prácticas de gestión de proyectos, pero Scrum en sí mismo no proporciona orientación suficiente a nivel técnico para equipos o a nivel de cartera/gobernanza para la dirección. Entonces, ¿por dónde empezamos?

¡Sea como ellos!

Con frecuencia, oímos a los jefes de desarrollo de software decir a sus empleados: "Deberíamos operar como Netflix" (u otra de esas empresas líderes). Por supuesto, podría hacerlo. Primero, hágase con varios cientos de miles de servidores y decenas de millones de usuarios...

El objetivo real

Por supuesto, el objetivo no es "hacer Scrum", "hacer desarrollo ágil", "hacer Lean" o lo que sea. El objetivo es estar en una posición que permita entregar software funcional que dé a los usuarios alguna nueva capacidad en cualquier momento. No dentro de unas semanas, unos meses o unos años, sino ahora. Para muchos equipos y organizaciones, la entrega continua parece una meta elevada e inalcanzable, sobre todo si les endosan un proceso que restringe la entrega a meses o semanas. Pero, como ocurre con cualquier objetivo, la clave es seguir apuntando en la dirección adecuada (véase la figura 9.1).

Si entrega en años, intente acortar el ciclo a meses. De meses, acórtelo a semanas. De un *sprint* de cuatro semanas, pruebe a reducir a dos. De un *sprint* de dos semanas, pruebe a pasar a una. Después, a diario y, por último, cuando se le pida. Tenga

en cuenta que ser capaz de entregar bajo demanda no significa que esté obligado a entregar cada minuto del día. Entregue cuando los usuarios lo necesiten, cuando hacerlo tiene sentido para el negocio.

Truco 88. Entregue cuando los usuarios lo necesiten.

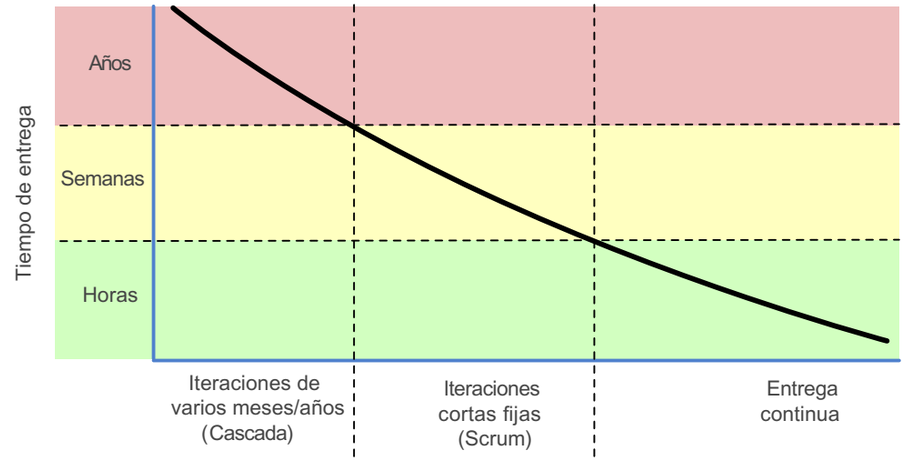

Figura 9.1.

Para poder pasar a este estilo de desarrollo continuo, necesita una infraestructura sólida como una roca, de la que hablaremos en el siguiente tema, "Kit pragmático básico". El desarrollo se hace en el tronco principal del sistema de control de versiones, no en ramas, y se utilizan técnicas como los *feature switches* para pasar características de prueba a los usuarios de manera selectiva.

Una vez que tenga la infraestructura en orden, necesita decidir cómo organizar el trabajo. Puede que los principiantes quieran empezar con Scrum para la gestión de proyectos, más las prácticas técnicas de la Programación Extrema (XP). Los equipos más disciplinados y experimentados pueden preferir técnicas Lean y Kanban, tanto para el equipo como para, tal vez, cuestiones de gobernanza más amplias.

Pero no se lo crea solo porque lo decimos nosotros. Investigue y pruebe estos enfoques usted mismo, pero tenga cuidado de no pasarse. Invertir en exceso en una metodología particular puede hacer que pase por alto las alternativas. Se acostumbrará a ella. Pronto le resultará difícil ver cualquier otra opción. Se anquilosará y ya no podrá adaptarse con rapidez. Para eso, le daría igual utilizar cocos.

significa que el resultado de una función puede convertirse en la entrada de otra. Si no tiene cuidado, hacer un cambio en el formato de los datos que genera una función puede causar un fallo en alguna otra parte más adelante en el flujo de transformaciones. Los lenguajes con buenos sistemas de tipos pueden ayudar a paliar esto.

Respuesta ejercicio 3

¡La baja tecnología al rescate! Haga algunos dibujos con el rotulador en la pizarra: un coche, un teléfono y una casa. No hace falta que sean obras de arte; el contorno con palitos sirve. Ponga notas en *post-its* que describan los contenidos de las páginas de destino en las áreas en las que se pueda hacer clic. A medida que progrese la reunión, puede perfeccionar los dibujos y las ubicaciones de los *post-its*.

Respuesta ejercicio 4

Puesto que queremos que el lenguaje sea extensible, haremos que el analizador esté gestionado por una tabla. Cada entrada de la tabla contiene la letra del comando, una bandera para indicar si se requiere un argumento y el nombre de la rutina a la que hay que llamar para manejar ese comando en particular.

lang/turtle.c

```
typedef struct {
  char  cmd;              /* la letra del comando */
  int hasArg;             /* ¿toma un argumento? */
  void (*func)(int, int); /* rutina a la que hay que llamar */
} Command;

static Command cmds[] = {
  { 'P',  ARG,     doSelectPen },
  { 'U',  NO_ARG,  doPenUp },
  { 'D',  NO_ARG,  doPenDown },
  { 'N',  ARG,     doPenDir },
  { 'E',  ARG,     doPenDir },
  { 'S',  ARG,     doPenDir },
  { 'W',  ARG,     doPenDir }
};
```

El programa principal es bastante simple: leer una línea, buscar el comando, tomar el argumento si es necesario y, después, llamar a la función manejadora.

lang/turtle.c

```
while (fgets(buff, sizeof(buff), stdin)) {

  Command *cmd = findCommand(*buff);

  if (cmd) {
    int   arg = 0;
    if (cmd->hasArg && !getArg(buff+1, &arg)) {
      fprintf(stderr, "'%c' needs an argument\n", *buff);
      continue;
    }

    cmd->func(*buff, arg);
  }
}
```


La función que busca un comando realiza una búsqueda lineal de la tabla, devolviendo la entrada correspondiente o NULL.

lang/turtle.c

```
Command *findCommand(int cmd) {
  int i;

  for (i = 0; i < ARRAY_SIZE(cmds); i++) {
    if (cmds[i].cmd == cmd)
      return cmds + i;
  }

  fprintf(stderr, "Unknown command '%c'\n", cmd);
  return 0;
}
```

Por último, leer el argumento numérico es bastante sencillo usando `sscanf`.

lang/turtle.c

```
int getArg(const char *buff, int *result) {
  return sscanf(buff, "%d", result) == 1;
}
```

Respuesta ejercicio 5

En realidad, ya ha resuelto este problema en el ejercicio anterior, donde ha escrito un intérprete para el lenguaje externo, contendrá el intérprete interno. En el caso de nuestro código de muestra, esto son las funciones `doXxx`.

Respuesta ejercicio 6

Usando BNF, una especificación de tiempo podría ser

tiempo ::= *hora ampm* | *hora* : *minuto ampm* | *hora* : *minuto*
ampm ::= am | pm
hora ::= *dígito* | *dígito dígito*
minuto ::= *dígito dígito*
dígito ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

"Uno de los libros más significativos de mi vida".
—**OBIE FERNANDEZ**, autor de *The Rails Way*

"Hace veinte años, la primera edición de *El programador pragmático* cambió por completo la trayectoria de mi carrera. Esta edición podría hacer lo mismo por la suya".
—**MIKE COHN**, autor de *Succeeding with Agile, Agile Estimating and Planning* y *User Stories Applied*

"... lleno de consejos prácticos, tanto técnicos como profesionales, que le servirán a usted y a sus proyectos durante muchos años".
—**ANDREA GOULET**, CEO de Corgibytes y fundadora de LegacyCode.Rocks

"... un rayo puede caer dos veces en el mismo sitio, y este libro es la prueba".
—**VM (VICKY) BRASSEUR**, directora de Open Source Strategy, Juniper Networks

El programador pragmático es uno de esos raros casos de libros técnicos que se leen, se releen y se vuelven a leer durante años. Tanto si es nuevo en el campo como si es un profesional experimentado, acabará encontrando ideas nuevas cada vez.

Dave Thomas y **Andy Hunt** escribieron la primera edición de este libro tan influyente en 1999 para ayudar a sus clientes a crear software mejor y a redescubrir el placer de escribir código. Estas lecciones han ayudado a una generación de programadores a examinar la propia esencia del desarrollo de software, independientemente de cualquier lenguaje, *framework* o metodología en particular, y la filosofía pragmática ha sido el germen de cientos de libros, *screencasts* y audiolibros, además de miles de carreras e historias de éxito.

Ahora, más de 20 años después, esta nueva edición reexamina lo que significa ser un programador moderno. Los temas abarcan desde la responsabilidad personal y el desarrollo profesional hasta técnicas de arquitectura para mantener su código flexible y fácil de adaptar y reutilizar. Lea este libro y podrá:

- Luchar contra la pudrición del software.
- Aprender continuamente.
- Evitar la trampa de la duplicación de la información.
- Escribir código flexible, dinámico y adaptable.
- Aprovechar el potencial de herramientas básicas.
- Evitar programar por casualidad.
- Aprender requisitos reales.
- Protegerse de las vulnerabilidades de seguridad.
- Resolver problemas subyacentes de código concurrente.
- Crear equipos de programadores pragmáticos.
- Responsabilizarse de su trabajo y de su carrera.
- Hacer pruebas de forma implacable y efectiva, incluyendo pruebas basadas en propiedades.
- Implementar el kit pragmático básico.
- Satisfacer a sus usuarios.

Escrito como una serie de secciones independientes y repleto de anécdotas clásicas y nuevas, ejemplos razonados y analogías interesantes, este libro ilustra los mejores enfoques y los mayores obstáculos de muchos aspectos diferentes del desarrollo de software. No importa si es un novato, un programador con experiencia o un director responsable de proyectos de software, utilice estas lecciones en su día a día y pronto observará mejorías en su productividad personal, exactitud y satisfacción laboral. Aprenderá las habilidades y desarrollará los hábitos y actitudes que forman los cimientos del éxito a largo plazo en su carrera.
Se convertirá en un programador pragmático.

Imagen e ilustración de cubierta: © 2022 Mihalec/idimair/
iStockphoto L.P./Guetty Images

Categoría: Ingeniería de software

ISBN 978-84-415-4587-8

2315176

ANAYA MULTIMEDIA

www.anayamultimedia.es