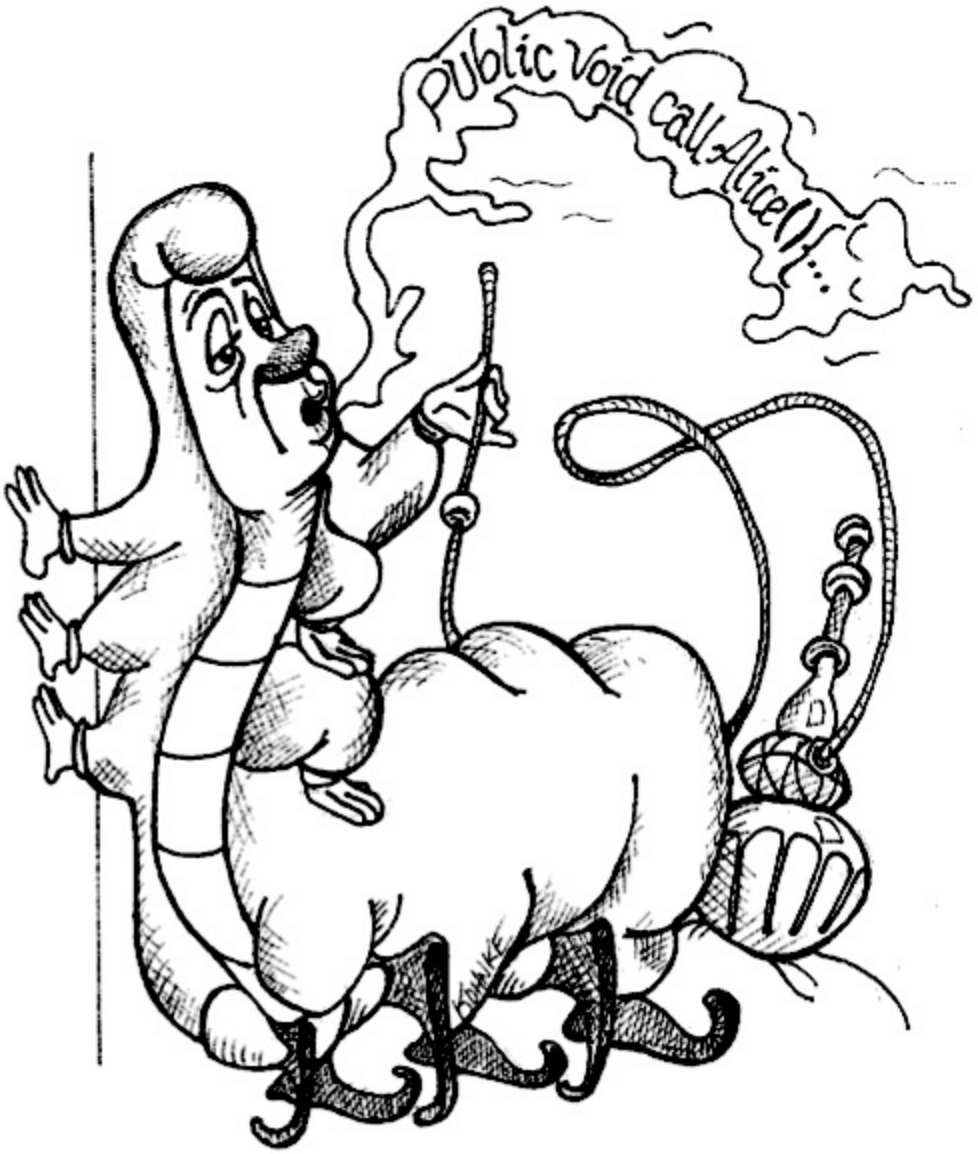
3
Funciones



En los inicios de la programación, creábamos sistemas a partir de rutinas y subrutinas. Después, en la época de Fortran y PL/1, creábamos nuestros sistemas con programas, subprogramas y funciones. En la actualidad, sólo las funciones han sobrevivido. Son la primera línea organizativa en cualquier programa. En este capítulo veremos cómo crearlas.

Fíjese en el código del Listado 3-1. Es complicado encontrar una función extensa en FitNesse[10], pero acabé encontrando ésta. No sólo es extensa, sino que también contiene código duplicado, muchas cadenas y tipos de datos extraños, además de API poco habituales y nada evidentes. Intente

comprenderlo en los próximos tres minutos.

Listado 3-1
HtmlUtil.java (FitNesse 20070619).

```
public static String testableHtml {
   PageData pageData,
   boolean includeSuiteSetup
} throws Exception {
   WikiPage wikiPage = pageData.getWikiPage();
   StringBuffer buffer = new StringBuffer();
   if (pageData.hasAttribute("Test")) {
     if (includeSuiteSetup) {
       WikiPage suiteSetup =
         PageCrawlerImpl.getInheritedPage(
           SuiteResponder.SUITE_SETUP_NAME, wikiPage
         );
       if (suiteSetup != null) {
         WikiPagePath pagePath =
           suiteSetup.getPageCrawler().getFullPath (suiteSetup);
         String pagePathName = PathParser.render(pagePath);
         buffer.append("!include -setup .")
           .append(pagePathName)
           .append("\n");
       }
     }
     WikiPage setup =
       PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
     if (setup != null) {
       WikiPagePath setupPath =
         wikiPage.getPageCrawler().getFullPath(setup);
       String setupPathName = PathParser.render(setupPath);
       buffer.append("!include -setup .")
         .append(setupPathName)
         .append("\n");
     }
   }
   buffer.append(pageData.getContent());
   if (pageData.hasAttribute("Test")) {
     WikiPage teardown =
       PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
     if (teardown != null) {
       WikiPagePath tearDownPath =
         wikiPage.getPageCrawler().getFullPath(teardown);
       String tearDownPathName = PathParser.render(tearDownPath);
       buffer.append("\n")
         .append("!include -teardown .")
         .append(tearDownPathName)
         .append("\n");
     }
     if (includeSuiteSetup) {
```

```
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
          SuiteResponder.SUITE_TEARDOWN_NAME,
          wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
          .append(pagePathName)
          .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

¿Tras tres minutos entiende la función? Seguramente no. Pasan demasiadas cosas y hay demasiados niveles de abstracción diferentes. Hay cadenas extrañas e invocaciones de funciones mezcladas en instrucciones `if` doblemente anidadas controladas por indicadores. Sin embargo, con sencillas extracciones de código, algún cambio de nombres y cierta reestructuración, pude capturar la intención de la función en las nueve líneas del Listado 3-2. Compruebe si ahora la entiende.

Listado 3-2
HtmlUtil.java (refactorización).

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages (testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

A menos que sea un alumno de FitNesse, seguramente no entienda los

detalles. Entenderá que la función se encarga de añadir páginas de configuración y detalles en una página de prueba, que después muestra en HTML. Si está familiarizado con JUnit[11], verá que esta función pertenece a algún tipo de estructura de pruebas basada en la Web y, evidentemente, es correcto. Resulta sencillo adivinar esta información del Listado 3-2 pero no del Listado 3-1. ¿Qué tiene la función del Listado 3-2 para que resulte sencilla de leer y entender? ¿Qué hay que hacer para que una función transmita su intención? ¿Qué atributos podemos asignar a nuestras funciones para que el lector pueda intuir el tipo de programa al que pertenecen?

Tamaño reducido

La primera regla de las funciones es que deben ser de tamaño reducido. La segunda es que *deben ser todavía más reducidas*. No es una afirmación que pueda justificar. No puedo mostrar referencias a estudios que demuestren que las funciones muy reducidas sean mejores. Lo que sí puedo afirmar es que durante casi cuatro décadas he creado funciones de diferentes tamaños. He creado monstruos de casi 3000 líneas y otras muchas funciones de entre 100 y 300 líneas. También he creado funciones de 20 a 30 líneas de longitud. Esta experiencia me ha demostrado, mediante ensayo y error, que las funciones deben ser muy reducidas.

En la década de 1980 se decía que una función no debía superar el tamaño de una pantalla. Por aquel entonces, las pantallas VT100 tenían 24 líneas por 80 columnas, y nuestros editores usaban 4 líneas para tareas administrativas. En la actualidad, con una fuente mínima y un monitor de gran tamaño, se pueden encajar 150 caracteres por línea y 100 líneas o más en una pantalla. Las líneas no deben tener 150 caracteres. Las funciones no deben tener 100 líneas de longitud. Las funciones deben tener una longitud aproximada de 20 líneas.

¿Qué tamaño mínimo debe tener una función? En 1999 visité a Kent Beck en su casa de Oregon. Nos sentamos y comenzamos a programar. Me enseñó un atractivo programa de Java/Swing que había llamado *Sparkle*. Generaba un efecto visual en pantalla, similar a la varita mágica del hada de

Cenicienta. Al mover el ratón, salían estrellitas del cursor, y descendían a la parte inferior de la pantalla en un campo gravitatorio simulado. Cuando Kent me enseñó el código, me sorprendió la brevedad de las funciones. Estaba acostumbrado a ver programas de Swing con funciones que ocupaban kilómetros de espacio vertical. En este programa, las funciones tenían dos, tres o cuatro líneas de longitud. Todas eran obvias. Todas contaban una historia y cada una llevaba a la siguiente en un orden atractivo. ¡Así de breves deberían ser todas las funciones![12]

¿Qué tamaño mínimo deben tener sus funciones? Deberían ser más breves que las del Listado 3-2. De hecho, el Listado 3-2 debería reducirse como el Listado 3-3.

Listado 3-3
HtmlUtil.java (nueva refactorización).

```
public static String renderPageWithSetupsAndTeardowns(
   PageData pageData, boolean isSuite) throws Exception {
   if (isTestPage(pageData))
     includeSetupAndTeardownPages(pageData, isSuite);
   return pageData.getHtml();
}
```

Bloques y sangrado

Esto implica que los bloques en instrucciones `if`, `else`, `while` y similares deben tener una línea de longitud que, seguramente, sea la invocación de una función. De esta forma, no sólo se reduce el tamaño de la función, sino que también se añade valor documental ya que la función invocada desde el bloque puede tener un nombre descriptivo. También implica que las funciones no deben tener un tamaño excesivo que albergue estructuras anidadas. Por tanto, el nivel de sangrado de una función no debe ser mayor de uno o dos. Evidentemente, de esta forma las funciones son más fáciles de leer y entender.

Hacer una cosa

Es evidente que el Listado 3-1 hace más de una cosa. Crea búferes, obtiene páginas, busca páginas heredadas, añade cadenas antiguas y genera HTML. El Listado 3-1 está muy ocupado realizando varias tareas. Por su parte, el Listado 3-3 sólo hace una cosa: incluye configuraciones y detalles en páginas de prueba.

El siguiente consejo lleva vigente, de una u otra forma, durante más de 30 años:

LAS FUNCIONES SÓLO DEBEN HACER UNA COSA. DEBEN HACERLO BIEN Y DEBE SER LO ÚNICO QUE HAGAN.

El problema de esta afirmación es saber qué es una cosa. ¿El Listado 3-3 hace una cosa? Se podría pensar que hace tres:

1. Determinar si la página es una página de prueba.
2. En caso afirmativo, incluir configuraciones y detalles.
3. Representar la página en HTML.

¿Cuál será de las tres? ¿La función hace una o tres cosas? Los tres pasos de la función se encuentran un nivel de abstracción por debajo del nombre de la función. Podemos describir la función como un breve párrafo TO (PARA) [13]:

> *Para `renderPageWithSetupsAndTeardowns`, comprobamos si la página es de prueba y, en caso afirmativo, incluimos las configuraciones y los detalles. En ambos casos, la representamos en HTML.*

Si una función sólo realiza los pasos situados un nivel por debajo del nombre de la función, entonces hace una cosa. En definitiva, creamos funciones para descomponer conceptos más amplios (es decir, el nombre de

la función) en un conjunto de pasos en el siguiente nivel de abstracción. Es evidente que el Listado 3-1 contiene pasos en distintos niveles de abstracción, por lo que es obvio que hace más de una cosa. Incluso el Listado 3-2 tiene tres niveles de abstracción, como ha demostrado la capacidad de reducirlo, pero sería complicado reducir con sentido el Listado 3-3. Podríamos extraer la instrucción `if` en la función `includeSetupsAndTeardownsIfTestPage`, pero sólo reduciríamos el código sin cambiar el nivel de abstracción.

Por ello, otra forma de saber que una función hace más de una cosa es extraer otra función de la misma con un nombre que no sea una reducción de su implementación [G34].

Secciones en funciones

Fíjese en el Listado 4-7. Verá que la función `generatePrimes` se divide en secciones como declaraciones, inicializaciones y filtros. Es un síntoma evidente de que hace más de una cosa. Las funciones que hacen una sola cosa no se pueden dividir en secciones.

Un nivel de abstracción por función

Para que las funciones realicen «una cosa», asegúrese de que las instrucciones de la función se encuentran en el mismo nivel de abstracción. El Listado 3-1 incumple esta regla. Incluye conceptos a un elevado nivel de abstracción, como `getHtml();` otros se encuentran en un nivel intermedio, como `StringpagePathName = PathParser.render(pagePath)` y hay otros en un nivel especialmente bajo, como `.append("\n").`

La mezcla de niveles de abstracción en una función siempre resulta confusa. Los lectores no sabrán si una determinada expresión es un concepto esencial o un detalle. Peor todavía, si se mezclan detalles con conceptos esenciales, aumentarán los detalles dentro de la función.

Leer código de arriba a abajo: la regla descendente

El objetivo es que el código se lea como un texto de arriba a abajo[14]. Queremos que tras todas las funciones aparezcan las del siguiente nivel de abstracción para poder leer el programa, descendiendo un nivel de abstracción por vez mientras leemos la lista de funciones. Es lo que denomino la regla descendente.

Para decirlo de otra forma, queremos leer el programa como si fuera un conjunto de párrafos `TO`, en el que cada uno describe el nivel actual de abstracción y hace referencia a los párrafos `TO` posteriores en el siguiente nivel.

> *Para incluir configuraciones y detalles, incluimos configuraciones, después del contenido de la página de prueba, y por último los detalles.*
> *Para incluir las configuraciones, incluimos la configuración de suite si se trata de una suite, y después la configuración convencional.*
> *Para incluir la configuración de suite; buscamos la jerarquía principal de la página `SuiteSetUp` y añadimos una instrucción `include` con la ruta de dicha página.*
> *Para buscar la jerarquía principal…*

A los programadores les resulta complicado aprender esta regla y crear funciones en un único nivel de abstracción, pero es un truco importante. Es la clave para reducir la longitud de las funciones y garantizar que sólo hagan una cosa. Al conseguir que el código se lea de arriba a abajo, se mantiene la coherencia de los niveles de abstracción.

Fíjese en el Listado 3-7 del final del capítulo. Muestra la función `testableHtml` modificada de acuerdo a estos principios. Cada función presenta a la siguiente y se mantiene en un nivel de abstracción coherente.

Instrucciones Switch

Es complicado usar una instrucción `switch` de tamaño reducido[15]. Aunque sólo tenga dos casos, es mayor de lo que un bloque o función debería ser. También es complicado crear una instrucción `switch` que haga una sola cosa. Por su naturaleza, las instrucciones `switch` siempre hacen N cosas. Desafortunadamente, no siempre podemos evitar las instrucciones `switch` pero podemos asegurarnos de incluirlas en una clase de nivel inferior y de no repetirlas. Para ello, evidentemente, recurrimos al polimorfismo.

Fíjese en el Listado 3-4. Muestra una de las operaciones que pueden depender del tipo de empleado.

Listado 3-4
Payroll.java.

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType (
   switch (e.type) {
     case COMMISSIONED:
       return calculateCommissionedPay(e);
     case HOURLY:
       return calculateHourlyPay(e);
     case SALARIED:
       return calculateSalariedPay(e);
     default:
       throw new InvalidEmployeeType(e.type);
   }
}
```

Esta función tiene varios problemas. Por un lado, es de gran tamaño y cuando se añadan nuevos tipos de empleado, aumentará más. Por otra parte, hace más de una cosa. También incumple el Principio de responsabilidad única (*Single Responsibility Principie* o SRP)[16] ya que hay más de un motivo para cambiarla. Además, incumple el Principio de abierto/cerrado (*Open Closed Principle* u OCP)[17], ya que debe cambiar cuando se añadan nuevos tipos, pero posiblemente el peor de los problemas es que hay un número ilimitado de funciones que tienen la misma estructura.

Por ejemplo, podríamos tener:

```
isPayday(Employee e, Date date),
```

o

```
  deliverPay(Employee e, Date date),
```

o muchas otras, todas con la misma estructura.

La solución al problema (véase el Listado 3-5) consiste en ocultar la instrucción `switch` en una factoría abstracta[18] e impedir que nadie la vea. La factoría usa la instrucción `switch` para crear las instancias adecuadas de los derivados de Employee y las distintas funciones, como `calculatePay`, `isPayday` y `deliverPay`, se entregarán de forma polimórfica a través de la interfaz `Employee`.

Listado 3-5
Employee y Factory.

```
  public abstract class Employee {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
  }
```

```
  public interface EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
  }
```

```
  public class EmployeeFactoryImpl implements EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType
{
      switch (r.type) {
        case COMMISSIONED:
          return new CommissionedEmployee(r);
        case HOURLY:
          return new HourlyEmployee(r);
        case SALARIED:
          return new SalariedEmployee(r);
        default:
          throw new InvalidEmployeeType(r.type);
      }
    }
  }
```

Mi regla general para las instrucciones `switch` es que se pueden tolerar si sólo aparecen una vez, se usan para crear objetos polimórficos y se ocultan tras una relación de herencia para que el resto del sistema no las pueda ver [G23]. Evidentemente, cada caso es diferente y en ocasiones se puede

incumplir una o varias partes de esta regla.

Usar nombres descriptivos

En el Listado 3-7, hemos cambiado el nombre de la función de ejemplo de `testableHtml` a `SetupTeardownIncluder.render`. Es un nombre más apropiado ya que describe mejor el cometido de la función. También hemos asignado a los métodos privados un nombre descriptivo como `isTestable` o `includeSetupAndTeardownPages`. No hay que olvidar el valor de los nombres correctos. Recuerde el principio de Ward: «Sabemos que trabajamos con código limpio cuando cada rutina es más o menos lo que esperábamos». Para alcanzar este principio, gran parte del esfuerzo se basa en seleccionar nombres adecuados para pequeñas funciones que hacen una cosa. Cuanto más reducida y concreta sea una función, más sencillo será elegir un nombre descriptivo. No tema los nombres extensos. Un nombre descriptivo extenso es mucho mejor que uno breve pero enigmático. Use una convención de nombres que permita leer varias palabras en los nombres de las funciones y use esas palabras para asignar a la función un nombre que describa su cometido.

No tema dedicar tiempo a elegir un buen nombre. De hecho, debería probar con varios nombres y leer el código con todos ellos. Los IDE modernos como Eclipse o IntelliJ facilitan el cambio de nombres. Use uno de estos IDE y experimente con diferentes nombres hasta que encuentre uno que sea lo bastante descriptivo.

La elección de nombres descriptivos clarifica el diseño de los módulos y le permite mejorarlos. No es extraño que la búsqueda de nombres adecuados genere una reestructuración favorable del código. Sea coherente con los nombres. Use las mismas frases, sustantivos y verbos en los nombres de función que elija para los módulos. Pruebe, por ejemplo, con `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` e `includeSetupPage`. La estructura similar de estos nombres permite que la secuencia cuente una historia. En realidad, si ve la secuencia anterior, seguramente se pregunte qué ha pasado con

`includeTeardownPages`, `includeSuiteTeardownPage` e `includeTeardownPage`.

Argumentos de funciones

El número ideal de argumentos para una función es cero. Después uno (monádico) y dos (diádico). Siempre que sea posible, evite la presencia de tres argumentos (triádico). Más de tres argumentos (poliádico) requiere una justificación especial y no es muy habitual.

Los argumentos son complejos ya que requieren un gran poder conceptual. Por ello suelo evitarlos en los ejemplos. Fíjese en `StringBuffer`. Podríamos haberlo pasado como argumento en lugar de como variable de instancia, pero los lectores habrían tenido que interpretarlo cada vez que lo vieran. Al leer la historia que cuenta el módulo, `includeSetupPage()` es más sencillo de interpretar que `includeSetupPageInto(newPageContent)`. El argumento se encuentra en un nivel de abstracción diferente que el nombre de la función y nos obliga a conocer un detalle (`StringBuffer`) que no es especialmente importante en ese momento.

Los argumentos son todavía más complicados desde un punto de vista de pruebas. Imagine la dificultad de crear todos los casos de prueba para garantizar el funcionamiento de las distintas combinaciones de argumentos. Si no hay argumentos, todo es más sencillo. Si hay uno, no es demasiado difícil. Con dos argumentos el problema es más complejo. Con más de dos argumentos, probar cada combinación de valores adecuados es todo un reto. Los argumentos de salida son más difíciles de entender que los de entrada. Al leer una función, estamos acostumbrados al concepto de información añadida

a la función a través de argumentos y extraída a través de un valor devuelto. No esperamos que la información se devuelva a través de los argumentos. Por ello, los argumentos de salida suelen obligamos a realizar una comprobación doble.

Un argumento de salida es la mejor opción, después de la ausencia de argumentos. `SetupTeardownIncluder.render(pageData)` se entiende bien. Evidentemente, vamos a representar los datos en el objeto `pageData`.

Formas monádicas habituales

Hay dos motivos principales para pasar un solo argumento a una función. Puede que realice una pregunta sobre el argumento, como en `boolean fileExists("MyFile")`, o que procese el argumento, lo transforme en otra cosa y lo devuelva. Por ejemplo, `InputStream fileOpen("MyFile")` transforma un nombre de archivo `String` en un valor devuelto `InputStream`. Los usuarios esperan estos dos usos cuando ven una función. Debe elegir nombres que realicen la distinción con claridad y usar siempre ambas formas en un contexto coherente (consulte el apartado sobre separación de consultas de comandos).

Una forma menos habitual pero muy útil para un argumento es un evento. En esta forma, hay argumento de entrada pero no de salida. El programa debe interpretar la invocación de la función como evento y usar el argumento para alterar el estado del sistema, por ejemplo, `void passwordAttemptFailedNtimes(int attempts)`. Use esta forma con precaución. Debe ser claro para el lector que se trata de un evento. Elija nombres y contextos con atención. Intente evitar funciones monádicas que no tengan estas formas, por ejemplo, `void includeSetupPageInto(StringBuffer pageText)`. El uso de un argumento de salida en lugar de un valor devuelto para realizar transformaciones resulta confuso. Si una función va a transformar su argumento de entrada, la transformación debe aparecer como valor devuelto. Sin duda `StringBuffertransform(StringBuffer in)` es mejor que `void`

`transform(StringBuffer out)`, aunque la implementación del primer caso devuelva solamente el argumento de entrada. Al menos se ajusta a la forma de la transformación.

Argumentos de indicador

Los argumentos de indicador son horribles. Pasar un valor Booleano a una función es una práctica totalmente desaconsejable. Complica inmediatamente la firma del método e indica que la función hace más de una cosa. Hace algo si el indicador es `true` y otra cosa diferente si es `false`. En el Listado 3-7 no se puede evitar, porque los invocadores ya pasan el indicador y el objetivo era limitar el ámbito a la función y después, pero la invocación de `render (true)` es confusa para el lector. Si se desplaza el ratón sobre la invocación vemos que `render (boolean isSuite)` puede ayudar, pero no demasiado. Tendremos que dividir la función en dos: `renderForSuite()` y `renderForSingleTest()`.

Funciones diádicas

Una función con dos argumentos es más difícil de entender que una función monádica. Por ejemplo `writeField(name)` es más fácil de entender que `writeField (outputStream, name)`[19]. Aunque en ambos casos el significado es evidente, la primera se capta mejor visualmente. La segunda requiere una breve pausa hasta que ignoramos el segundo parámetro, lo que en última instancia genera problemas ya que no debemos ignorar esa parte del código. Las partes que ignoramos son las que esconden los errores. Pero en ocasiones se necesitan dos argumentos. Por ejemplo. `Point p = new Point(0,0);` es totalmente razonable. Los puntos cartesianos suelen adoptar dos argumentos. De hecho, sería muy sorprendente ver `Point(0)`. Sin embargo, en este caso ambos argumentos son componentes ordenados de un mismo valor, mientras que `outputStream` y `name` carecen de una cohesión o un orden natural.

Incluso funciones diádicas evidentes como `assertEquals(expected, actual)` resultan problemáticas. ¿Cuántas veces ha incluido el valor real en su posición esperada? Los dos argumentos carecen de un orden natural. El orden real y esperado es una convención que se adquiere gracias a la práctica.

Las combinaciones diádicas no son el mal en persona y tendrá que usarlas. Sin embargo, recuerde que tienen un precio y que debe aprovechar los mecanismos disponibles para convertirlas en unitarias. Por ejemplo, puede hacer que el método `writeField` sea un miembro de `outputStream` para poder usar `outputStream.writeField(name)`, o podría convertir `outputStream` en una variable miembro de la clase actual para no tener que pasarla. Incluso podría extraer una nueva clase como `FieldWriter` que usara `outputStream` en su constructor y tuviera un método `write`.

Triadas

Las funciones que aceptan tres argumentos son sin duda mucho más difíciles de entender que las de dos. Los problemas a la hora de ordenar, ignorar o detenerse en los argumentos se duplican. Piense atentamente antes de crear una triada.

Por ejemplo, fíjese en la sobrecarga de `assertEquals` que acepta tres argumentos: `assertEquals(message, expected, actual)`. ¿Cuántas veces lee el mensaje y piensa que es lo esperado? He visto esta triada en concreto muchas veces. De hecho, siempre que la veo, tengo que repasarla antes de ignorar el mensaje.

Por otra parte, hay otra triada que no es tan negativa: `assertEquals(1.0, amount, .001)`. Aunque también exija doble atención, merece la pena. Conviene recordar siempre que la igualdad de los valores de coma flotante es algo relativo.

Objeto de argumento

Cuando una función parece necesitar dos o más argumentos, es probable que

alguno de ellos se incluya en una clase propia. Fíjese en la diferencia entre las dos siguientes declaraciones:

```
Circle makeCircle (double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

La reducción del número de argumentos mediante la creación de objetos puede parecer una trampa pero no lo es. Cuando se pasan grupos de variables de forma conjunta, como x e y en el ejemplo anterior, es probable que formen parte de un concepto que se merece un nombre propio.

Listas de argumentos

En ocasiones tendremos que pasar un número variable de argumentos a una función. Fíjese en el método `String.format`:

```
String.format (“%s worked %.2f hours.”, name, hours);
```

Si los argumentos variables se procesan de la misma forma, como en el ejemplo anterior, serán equivalentes a un único argumento de tipo `List`. Por tanto, `String.format` es en realidad diádico. De hecho, la siguiente declaración de `String.format` es claramente diádica.

```
public String format(String format, Object… args)
```

Así pues, se aplican las mismas reglas. Las funciones que aceptan argumentos variables pueden ser monádicas, diádicas o incluso triádicas, pero sería un error asignar más argumentos.

```
void monad(Integer… args);
void dyad(String name, Integer… args);
void triad(String name, int count, Integer… args);
```

Verbos y palabras clave

La selección de nombres correctos para una función mejora la explicación de su cometido, así como el orden y el cometido de los argumentos. En formato monádico, la función y el argumento deben formar un par de verbo y sustantivo. Por ejemplo, `write(name)` resulta muy evocador. Sea lo que sea `name`, sin duda se escribe `(write)`.

Un nombre más acertado podría ser `writeField(name)`, que nos dice que `name` es un campo `(field)`. Éste es un ejemplo de palabra clave como nombre de función. Con este formato codificamos los nombres de los argumentos en el nombre de la función. Por ejemplo, `assertEquals` se podría haber escrito como `assertExpectedEqualsActual(expected, actual)`, lo que mitiga el problema de tener que recordar el orden de los argumentos.

Sin efectos secundarios

Los efectos secundarios son mentiras. Su función promete hacer una cosa, pero también hace otras cosas ocultas. En ocasiones realiza cambios inesperados en las variables de su propia clase. En ocasiones las convierte en las variables pasadas a la función o a elementos globales del sistema. En cualquier caso, se comete un engaño que suele provocar extrañas combinaciones temporales y dependencias de orden.

Fíjese en la función del Listado 3-6, aparentemente inofensiva. Usa un algoritmo estándar para comparar `userName` con `password`. Devuelve `true` si coinciden y `false` si hay algún problema, pero también hay un efecto secundario. ¿Lo detecta?

Listado 3-6
UserValidator.java.

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if (“Valid Password”.equals(phrase)){
        Session.initialize();
        return true;
      }
    }
    return false;
  }
```

```
  }
```

El efecto secundario es la invocación de `Session.initialize()`. La función `checkPassword`, por su nombre, afirma comprobar la contraseña. El nombre no implica que inicialice la sesión. Por tanto, un invocador que se crea lo que dice el nombre de la función se arriesga a borrar los datos de sesión actuales cuando decida comprobar la validez del usuario. Este efecto secundario genera una combinación temporal. Es decir, sólo se puede invocar `checkPassword` en determinados momentos (cuando se pueda inicializar la sesión). Si no se invoca en orden, se pueden perder los datos de la sesión. Las combinaciones temporales son confusas, en especial cuando se ocultan como efecto secundario. Si tiene que realizar una combinación temporal, hágalo de forma clara en el nombre de la función. En este caso, podríamos cambiar el nombre de la función por `checkPasswordAndInitializeSession`, pero incumpliría la norma de hacer una sola cosa.

Argumentos de salida

Los argumentos suelen interpretarse como entradas de una función. Si lleva varios años programando, estoy seguro de que habrá visto un argumento que en vez de ser de entrada era de salida. Por ejemplo;

```
  appendFooter(s);
```

¿Está función añade `s` al final de algo? ¿O añade el final de algo a `s`? ¿`s` es una entrada o una salida? Lo sabemos al ver la firma de la función:

```
  public void appendFooter(StringBuffer report)
```

Esto lo aclara todo, pero para ello hay que comprobar la declaración de la función. Todo lo que le obligue a comprobar la firma de la función es un esfuerzo doble. Es una pausa cognitiva y debe evitarse.

Antes de la programación orientada a objetos, era necesario tener argumentos de salida. Sin embargo, gran parte de su necesidad desaparece en los lenguajes orientados a objetos, pensados para actuar como argumento de salida. Es decir, sería más indicado invocar `appendFooter` como `report.appendFooter();`.

Por lo general, los argumentos de salida deben evitarse. Si su función

tiene que cambiar el estado de un elemento, haga que cambie el estado de su objeto contenedor.

Separación de consultas de comando

Las funciones deben hacer algo o responder a algo, pero no ambas cosas. Su función debe cambiar el estado de un objeto o devolver información sobre el mismo, pero ambas operaciones causan confusión. Fíjese en la siguiente función:

```
public boolean set(String attribute, String value);
```

Esta función establece el valor de un atributo y devuelve `true` en caso de éxito o `false` si el atributo no existe. Esto provoca la presencia de una extraña instrucción como la siguiente:

```
if (set(“username”, “unclebob”))…
```

Imagínelo desde el punto de vista del lector. ¿Qué significa? ¿Pregunta si el atributo «`username`» se ha establecido antes en «`unclebob`», o si el atributo «username» se ha establecido correctamente en «`unclebob`»? Es complicado saberlo por la invocación ya que no es evidente si set es un verbo o un adjetivo.

El autor pretendía que `set` fuera un verbo, pero el contexto de la instrucción `if` parece un adjetivo. La instrucción se lee como «si el atributo `username` se ha establecido previamente en `unclebob`», no como «establecer el atributo `username` en `unclebob` y si funciona, entonces…». Podríamos solucionarlo si cambiamos el nombre de la función `set` por `setAndCheckIfExists`, pero no mejoraría la legibilidad de la instrucción `if`. La verdadera solución es separar el comando de la consulta para evitar la ambigüedad.

```
if (attributeExists(“username”)) {
   setAttribute(“username”, “unclebob”);
   …
}
```

Mejor excepciones que devolver códigos de error

Devolver códigos de error de funciones de comando es un sutil incumplimiento de la separación de comandos de consulta. Hace que los comandos usados asciendan a expresiones en los predicados de las instrucciones `if`.

```
if (deletePage(page) == E_OK)
```

No padece la confusión entre verbo y adjetivo, pero genera estructuras anidadas. Al devolver un código de error se crea un problema: el invocador debe procesar el error de forma inmediata.

```
if (deletePage(page) == E_OK) {
   if (registry.deleteReference(page.name) == E_OK) {
     if (configKeys.deleteKey(page.name.makeKey()) == E_OK) {
       logger.log(“page deleted”);
     } else {
       logger.log(“configKey not deleted”);
     }
   } else {
     logger.log(“deleteReference from registry failed”);
   }
} else {
   logger.log(“delete failed”);
   return E_ERROR;
}
```

Por otra parte, si usa excepciones en lugar de códigos de error, el código de procesamiento del error se puede separar del código de ruta y se puede simplificar:

```
try {
   deletePage(page);
   registry.deleteReference(page.name);
   configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
   logger.log(e.getMessage());
}
```

Extraer bloques Try/Catch

Los bloques `try/catch` no son atractivos por naturaleza. Confunden la estructura del código y mezclan el procesamiento de errores con el normal.

Por ello, conviene extraer el cuerpo de los bloques `try` y `catch` en funciones individuales.

```
public void delete(Page page) {
   try {
     deletePageAndAllReferences(page);
   }
   catch (Exception e) {
     logError(e);
   }
}

private void deletePageAndAllReferences(Page page) throws Exception {
   deletePage(page);
   registry.deleteReference(page.name);
   configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
   logger.log(e.getMessage());
}
```

En este caso, la función `delete` es de procesamiento de errores. Es fácil de entender e ignorar. La función `deletePageAndAllReferences` es para los procesos de borrar una página. El procesamiento de errores se puede ignorar. De este modo, la separación facilita la comprensión y la modificación del código.

El procesamiento de errores es una cosa

Las funciones sólo deben hacer una cosa y el procesamiento de errores es un ejemplo. Por tanto, una función que procese errores no debe hacer nada más. Esto implica (como en el ejemplo anterior) que, si una función incluye la palabra clave `try`, debe ser la primera de la función y que no debe haber nada más después de los bloques `catch/finally`.

El imán de dependencias Error.java

La devolución de códigos de error suele implicar que existe una clase o enumeración en la que se definen los códigos de error.

```
public enum Error {
```

```
    OK,
    INVALID,
    NO_SUCH,
    LOCKED,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
}
```

Clases como ésta son un *imán para las dependencias*; otras muchas clases deben importarlas y usarlas. Por ello, cuando cambia la enumeración `Error`, es necesario volver a compilar e implementar dichas clases[20]. Esto añade presión a la clase `Error`. Los programadores no quieren añadir nuevos errores porque tendrán que volver a generar e implementarlo todo. Por ello, reutilizan códigos de error antiguos en lugar de añadir otros nuevos.

Al usar excepciones en lugar de códigos de error, las nuevas excepciones son derivaciones de la clase de la excepción. Se pueden añadir sin necesidad de volver a compilar o implementar[21].

No repetirse[22]

Fíjese de nuevo en el Listado 3-1; verá que hay un algoritmo que se repite cuatro veces, en los casos `SetUp`, `SuiteSetUp`, `TearDown` y `SuiteTearDown`. No es fácil detectar esta repetición ya que las cuatro instancias se mezclan con otro código, pero la duplicación es un problema ya que aumenta el tamaño del código y requerirá una modificación cuádruple si alguna vez cambia el algoritmo.

También se cuadriplica el riesgo de errores.

Esta duplicación se remedia gracias al método `include` del Listado 3-7. Vuelva a leer el código y fíjese en cómo se ha mejorado la legibilidad del código reduciendo la duplicación.

La duplicación puede ser la raíz de todos los problemas del *software*. Existen numerosos principios y prácticas para controlarla o eliminarla. Imagine que todas las formas normales de la base de datos de Codd sirvieran

para eliminar la duplicación de datos. Imagine también cómo la programación orientada a objetos concentra el código en clases base que en otros casos serian redundantes. La programación estructurada, la programación orientada a aspecto y la orientada a componentes son, en parte, estrategias para eliminar duplicados. Parece que, desde la aparición de las subrutinas, las innovaciones en desarrollo de *software* han sido un intento continuado por eliminar la duplicación de nuestro código fuente.

Programación estructurada

Algunos programadores siguen las reglas de programación estructurada de Edsger Dijkstra[23]. Dijkstra afirma que todas las funciones y todos los bloques de una función deben tener una entrada y una salida. Estas reglas implican que sólo debe haber una instrucción `return` en una función, que no debe haber instrucciones `break` o `continue` en un bucle y nunca, bajo ningún concepto, debe haber instrucciones `goto`.

Aunque apreciemos los objetivos y disciplinas de la programación estructurada, no sirven de mucho cuando las funciones son de reducido tamaño. Su verdadero beneficio se aprecia en funciones de gran tamaño.

Por tanto, si sus funciones son de tamaño reducido, una instrucción `return`, `break` o `continue` no hará daño alguno y en ocasiones puede resultar más expresiva que la regla de una entrada y una salida. Por otra parte, `goto` sólo tiene sentido en funciones de gran tamaño y debe evitarse.

Cómo crear este tipo de funciones

La creación de *software* es como cualquier otro proceso creativo. Al escribir un informe o un artículo, primero se estructuran las ideas y después el mensaje hasta que se lea bien. El primer borrador puede estar desorganizado, de modo que lo retoca y mejora hasta que se lea de la forma adecuada.

Cuando creo funciones, suelen ser extensas y complicadas, con abundancia de sangrados y bucles anidados. Con extensas listas de

argumentos, nombres arbitrarios y código duplicado, pero también cuento con una serie de pruebas de unidad que abarcan todas y cada una de las líneas de código.

Por tanto, retoco el código, divido las funciones, cambio los nombres y elimino los duplicados. Reduzco los métodos y los reordeno. En ocasiones, elimino clases enteras, mientras mantengo las pruebas.

Al final, consigo funciones que cumplen las reglas detalladas en este capítulo. No las escribo al comenzar y dudo que nadie pueda hacerlo.

Conclusión

Todo sistema se crea a partir de un lenguaje específico del dominio diseñado por los programadores para describir dicho sistema. Las funciones son los verbos del lenguaje y las clases los sustantivos. No es volver a la noción de que los sustantivos y verbos de un documento de requisitos son las clases y funciones de un sistema. Es una verdad mucho más antigua. El arte de la programación es, y ha sido siempre, el arte del diseño del lenguaje.

Los programadores experimentados piensan en los sistemas como en historias que contar, no como en programas que escribir. Recurren a las prestaciones del lenguaje de programación seleccionado para crear un lenguaje expresivo mejor y más completo que poder usar para contar esa historia. Parte de ese lenguaje es la jerarquía de funciones que describen las acciones que se pueden realizar en el sistema. Dichas acciones se crean para usar el lenguaje de dominio concreto que definen para contar su pequeña parte de la historia.

En este capítulo hemos visto la mecánica de la creación de funciones correctas. Si aplica estas reglas, sus funciones serán breves, con nombres correctos, y bien organizadas, pero no olvide que su verdadero objetivo es contar la historia del sistema y que las funciones que escriba deben encajar en un lenguaje claro y preciso que le sirva para contar esa historia.

SetupTeardownIncluder

Listado 3-7
SetupTeardownIncluder.java.

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
   private PageData pageData;
   private boolean isSuite;
   private WikiPage testPage;
   private StringBuffer newPageContent;
   private PageCrawler pageCrawler;

   public static String render(PageData pageData) throws Exception {
     return render(pageData, false);
   }

   public static String render(PageData pageData, boolean isSuite)
     throws Exception {
     return new SetupTeardownIncluder(pageData).render(isSuite);
   }

   private SetupTeardownIncluder(PageData pageData) {
     this.pageData = pageData;
     testPage = pageData.getWikiPage();
     pageCrawler = testPage.getPageCrawler();
     newPageContent = new StringBuffer();
   }

   private String render(boolean isSuite) throws Exception {
     this.isSuite = isSuite;
     if (isTestPage())
       includeSetupAndTeardownPages();
     return pageData.getHtml();
   }

   private boolean isTestPage() throws Exception {
     return pageData.hasAttribute("Test");
   }

   private void includeSetupAndTeardownPages() throws Exception {
     includeSetupPages();
     includePageContent();
     includeTeardownPages();
     updatePageContent();
   }

   private void includeSetupPages() throws Exception {
     if (isSuite)
       includeSuiteSetupPage();
     includeSetupPage();
   }
```

```
private void includeSuiteSetupPage() throws Exception {
  include(SuiteResponder.SUITE_SETUP_NAME, “-setup”);
}

private void includeSetupPage() throws Exception {
  include(“SetUp”, “-setup”);
}

private void includePageContent() throws Exception {
  newPageContent.append(pageData.getContent());
}

private void includeTeardownPages() throws Exception {
  includeTeardownPage();
  if (isSuite)
    includeSuiteTeardownPage();
}

private void includeTeardownPage() throws Exception {
  include(“TearDown”, “-teardown”);
}

private void includeSuiteTeardownPage() throws Exception {
  include(SuiteResponder.SUITE_TEARDOWN_NAME, “-teardown”);
}

private void updatePageContent() throws Exception {
  pageData.setContent(newPageContent.toString());
}

private void include(String pageName, String arg) throws Exception (
  WikiPage inheritedPage = findInheritedPage(pageName);
  if (inheritedPage != null) {
    String pagePathName = getPathNameForPage(inheritedPage);
    buildIncludeDirective(pagePathName, arg);
  }
}

private WikiPage findInheritedPage(String pageName) throws Exception {
  return PageCrawlerImpl.getInheritedPage(pageName, testPage);
}

private String getPathNameForPage(WikiPage page) throws Exception {
  WikiPagePath pagePath = pageCrawler.getFullPath(page);
  return PathParser.render(pagePath);
}

private void buildIncludeDirective(String pagePathName, String arg) {
  newPageContent
    .append(“\n!include ”)
    .append(arg)
    .append(“ .”)
    .append(pagePathName)
    .append(“\n”);
}
```

```
}
```

Bibliografía

- **[KP78]**: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.
- **[PPP02]**: Robert C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2002.
- **[GOF]**: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison Wesley, 1996.
- **[PRAG]**: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.
- **[SP72]**: *Structured Programming*, O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.