El camino a un mejor programador

Esteban Manchado Velázquez, Joaquín Caraballo Moreno, Yeray Darias Camacho

Índice general

[Licencia](#)
[Agradecimientos](#)
[Prólogo](#)
[Lecciones de aprender un lenguaje funcional](#)
[Documentación activa](#)
[Siete problemas al probar programas](#)
[Calidad en software](#)
[Integración continua](#)
[Desarrollo dirigido por pruebas](#)
[Bibliografía](#)

Licencia

Agradecimientos

A todos aquellos que ayudaron de una manera u otra en la creación de este libro, con consejos, apoyo o ideas. Las siguientes personas se merecen una mención especial:

- Sara Pettersson, por la portada
- Mario Hernández, por el prólogo
- Carlos Ble, por ayudar con la revisión de varios artículos y por sus ideas y consejos
- Juanje Ojeda, por sus ideas y empuje iniciales

Prólogo

Se cuenta que una vez le preguntaron a Miguel Ángel cómo había procedido para esculpir el David. Su respuesta fue que lo que había hecho era, simplemente, eliminar del bloque de piedra original todo lo que sobraba. Independientemente de la verosimilitud de esta anécdota, podemos encontrar en ella la enseñanza de una actitud a tomar para abordar problemas prácticos de desarrollo en ingeniería.

Si la pregunta se le hiciese a un escultor que modelase en barro la respuesta equivalente sería, posiblemente, que lo que había hecho era rellenar el espacio disponiendo el barro estrictamente necesario en las zonas adecuadas. Estas dos respuestas se refieren a dos aproximaciones para abordar procesos constructivos: la sustractiva, natural en la escultura en piedra, en la que se eliminan elementos de una solución primigenia hasta que se cumple con las restricciones de la solución definitiva, y la aditiva en la que se aborda el problema partiendo de una solución inicial vacía, añadiendo elementos según ciertos criterios con el fin de conseguir que la

solución final cumpla los objetivos de diseño. Estas dos son aproximaciones extrapolables a otras disciplinas y en general, en muchos otros casos, la solución se alcanza con aproximaciones mixtas, en las que se opera con acciones aditivas y sustractivas combinadas.

El desarrollo de software es una disciplina práctica relativamente nueva que tiene poco más de medio siglo de existencia, pero que conceptualmente es hija de las disciplinas clásicas constructivas englobadas bajo el paraguas de las ingenierías y la arquitectura. De ellas hereda una de las habilidades que necesita un buen informático; la de resolver problemas. Por ella me refiero a la capacidad, en palabras de Allen Downey, el autor de «Think Python. How to Think Like a Computer Scientist», para formular problemas, pensar en soluciones de manera creativa, y expresarlas de forma clara y precisa. A esto se debe probablemente mucha de la genialidad de Miguel Ángel.

Volviendo a las disciplinas clásicas de la ingeniería y la arquitectura, el objetivo de estas no es el de las ciencias clásicas de carácter analítico, es decir, no intenta entender, explicar y describir algo ya existente como lo hacen la física, la química o la biología, sino más bien sintético, es decir, orientado a desarrollar y construir cosas nuevas que satisfagan un objetivo expresado normalmente por la función definida a priori que debe cumplir el resultado. En este ámbito, el conocimiento acerca de cómo realizar bien las tareas de desarrollo de nuevas soluciones es un trabajo de destilación que se produce en ciclos de vida resumibles en «observación-análisis-propuestas de solución-realización práctica-evaluación de resultados-refinamiento, para alcanzar el nivel de aceptación-corrección para iniciar de nuevo el ciclo o descarte de la propuesta» según corresponda. Estos procesos se realizan iterativamente en el tiempo, constituyendo la fuente del conocimiento práctico del saber hacer de la disciplina y que se terminan expresando como recetas metodológicas y conjuntos de buenas prácticas para ejecutar los proyectos y tareas futuros. Su fin es llevar esos proyectos a buen término en forma de una solución que sea eficiente y efectiva y cumpla las restricciones impuestas al mínimo coste de realización posible medido en horas-hombre. Este proceso es análogo, de nuevo, a los desarrollos de la ingeniería y la arquitectura primigenia, donde los constructores y albañiles desarrollaban su conocimiento y metodologías de forma práctica enfrentándose a retos y problemas nuevos. Ello permitió, por ejemplo, construir las grandes obras de ingeniería civil durante la época de esplendor de Roma, como es el caso de los acueductos, palacios o coliseos, o la gran arquitectura revolucionaria de las catedrales de estilo gótico. Esas construcciones fueron productos pero también retos, que permitieron desarrollar conocimiento y metodologías que se explotaron y depuraron durante siglos.

La realización de proyectos en disciplinas sintéticas es pues una combinación virtuosa de la concepción y uso de las herramientas adecuadas para el desarrollo, con el conocimiento, las habilidades con esas herramientas y procesos y la experiencia disponible en la mente de quienes los ejecutan, que se manifiesta en la capacidad de resolución de problemas y se expresa como recetas metodológicas y conjuntos de buenas prácticas para llevar a buen fin los proyectos por parte de equipos de personas cualificadas. Una buena ingeniería o una buena arquitectura podrían entenderse en el sentido de Miguel Ángel como las disciplinas que tienen por objetivo el uso adecuado del conjunto de herramientas, conocimiento y habilidades combinados con una metodología que permite construir lo nuevo que se propone cumpliendo los objetivos de diseño, buscando quedarnos con lo estrictamente necesario en sentido constructivo y eliminando lo superfluo.

¿Cuál es la diferencia entre el desarrollo de proyectos de programación y el desarrollo de proyectos de ingeniería o arquitectura más o menos convencionales? Pues, conceptualmente el único conjunto de diferencias surge, básicamente, de la naturaleza del objeto de trabajo. En el caso de los últimos, el objeto lo constituyen tangibles físicos, mientras que en el caso del primero, el objeto es fundamentalmente un intangible: todo lo relacionado con la información y sus transformaciones. Además, por la naturaleza de su resultado y al contrario que en otras disciplinas constructivas, el objeto de la ejecución de proyectos software no está sometido a ley física alguna, excepto la que regula la entropía. De hecho, una gran parte de la actividad metodológica en el desarrollo de proyectos está dirigida a luchar contra la degradación y el desorden. Esta situación es la que establece la particularidad de la ejecución de proyectos software como cuerpo disciplinar, y es la que delimita el territorio sobre el que se tienen que desarrollar herramientas, conocimiento y metodologías.

Este libro pretende ser una aportación a la recopilación y la transmisión del conocimiento y la experiencia de un grupo de buenos y experimentados programadores profesionales en el desarrollo de proyectos de programación. Constituye pues un documento valioso para aquellas personas interesados en el mundo profesional del software.

Un valor interesante del libro reside en que se dedica a temas que no suelen formar parte del flujo de conocimientos que se manejan usualmente en los procesos de formación básica de los desarrolladores e ingenieros de software, como son los que aparecen en el índice de este documento: el desarrollo de software bajo el paradigma de programación funcional, la documentación activa, la prueba de programas, la calidad del software o los procesos de integración continua. Más bien podemos entenderlo como un texto que intenta servir como transmisor de conocimiento experto desde quienes tienen años de experiencia profesional en el campo y se dirige hacia un amplio espectro de informáticos, tanto experimentados en otros saberes, como noveles en estos temas. En este sentido constituye una obra de valor en los ámbitos que le compete.

Espero y confío en que esta contribución sirva para los fines que se propusieron sus autores y resulte de utilidad para sus lectores.

Mario Hernández
Catedrático de Universidad en Ciencias de la Computación e Inteligencia Artificial
Las Palmas de Gran Canaria, octubre de 2012

Lecciones de aprender un lenguaje funcional

Esteban Manchado Velázquez

Los lenguajes funcionales son una familia de lenguajes que la mayoría de los programadores conoce de oídas, pero desgraciadamente no muchos conocen suficientemente bien. Y digo «desgraciadamente» porque, independientemente de que por una razón u otra los lenguajes funcionales son muy poco demandados en el mercado laboral, aprenderlos nos puede brindar muchas ideas interesantes, patrones, buenas costumbres y lecciones que podemos aplicar a muchos otros lenguajes.

Es difícil trazar una línea clara entre los lenguajes funcionales y los lenguajes no funcionales, pero podemos citar Lisp, Haskell, Erlang, Scala y Clojure como los lenguajes funcionales más populares actualmente. Muchos lenguajes de programación populares tienen algunos rasgos funcionales, como Javascript, Ruby y Python. Los lenguajes funcionales, o en general, de cualquier paradigma al que no estemos acostumbrados, nos pueden dar ideas que podemos aplicar no solamente en estos lenguajes que tienen rasgos funcionales, sino en casi cualquier lenguaje.

Aprender un lenguaje funcional lo suficiente como para tener unas nociones e inspirarse, no tiene por qué llevar mucho tiempo. Además, no sólo disponemos de Internet, sino también de excelentes libros que están pensados precisamente para programadores que vienen de otros lenguajes. El resto de este artículo explora algunas técnicas, buenas costumbres e ideas comunes en lenguajes funcionales, que podemos aplicar fácilmente en otros lenguajes. Por supuesto, algunas de estas lecciones se pueden aprender simplemente por experiencia, y no son necesariamente exclusivas de los lenguajes funcionales.

Los lenguajes son diferentes

Los lenguajes de programación son como los idiomas humanos en muchos sentidos: tienen sintaxis, expresiones comunes, son mejores o peores que otros para expresar ciertas ideas, se pueden «hablar» con mayor o menor fluidez e incluso se puede decir que sus «hablantes» tienen una cierta «cultura» diferente de los «hablantes» de otros lenguajes.

Así, cuando aprendemos un lenguaje nuevo es un error verlo como «una nueva sintaxis»: aprender un lenguaje bien, nos hace cambiar cómo pensamos y cómo resolvemos los problemas. Por ejemplo, digamos que empezamos a aprender Lisp y que sólo conocemos lenguajes imperativos «clásicos» como C o Java. Como parte de un programa que estamos escribiendo, tenemos una función que calcula el valor total a pagar por unos artículos. Los parámetros de entrada son el importe de cada artículo, el número de artículos, el porcentaje de impuesto, y el límite pasado el cual hay que aplicar el impuesto (p.ej.: tenemos 2 artículos a 5 euros cada uno, un impuesto del 10% y un límite de 8 euros; el precio final será 11. Si el límite fuese 15 euros, el precio final sería 10 porque no se aplicaría el impuesto). Una forma de escribirla en Lisp sería la siguiente:

```
(defun calculate-price-BAD (cost nitems limit percent-tax)
  (setq total-price (* cost nitems))
  (if (> total-price limit)
      (progn
        (setq tax (* total-price (/ percent-tax 100)))
        (setq total-price (+ total-price tax))))
  total-price)
```

Mal ejemplo de cómo resolverlo en Lisp (calculate-prices-bad.lisp)

Sin embargo, si escribimos la función así no aprendemos nada nuevo y mostramos nuestro «acento extranjero» al escribir Lisp. Este código no es legible ni para programadores de Java ni para programadores de Lisp, y además no aprovechamos las ventajas del lenguaje, mientras que sufrimos sus desventajas. Simplemente usamos Lisp como «un mal Java». Compárese el anterior código con el siguiente:

```
(defun calculate-price (cost nitems limit percent-tax)
  (let* ((total-price (* cost nitems))
         (tax         (* total-price (/ percent-tax 100))))
    (if (> total-price limit)
      (+ total-price tax)
      total-price)))
```

Solución usando Lisp de una forma más convencional (calculate-prices.lisp)

Este código se parece más a lo que muchos programadores de Lisp esperarían o escribirían ellos mismos, y a la mayoría de las personas que sepan un poco de Lisp les parecerá más legible y fácil de mantener. La razón es que estamos jugando con las fortalezas de Lisp, en vez de intentar adaptar el lenguaje a nuestros conocimientos previos.

Concretamente, el segundo ejemplo se aprovecha de dos detalles muy comunes en Lisp que no son tan comunes en lenguajes no funcionales:

1. No usar variables, sino «valores con nombre». O, dicho de otra manera, evitar asignar más de una vez a cada variable. Esto se puede ver en el bloque `let*`: las dos «variables» del bloque, `total-price` y `tax`, nunca reciben otros valores. Así, el bloque `let*` contiene una lista de valores con nombres simbólicos, lo que hace el código más claro y mantenible.
2. La construcción `if` sigue el patrón funcional de ser una *expresión* que devuelve un valor. En este caso, recibe una condición y dos expresiones: la primera se devuelve si la condición es verdadera, y la segunda si la condición resulta ser falsa. Funciona de una forma parecida al operador ternario de C y Java (`valor = condicion ? valorsiverdadero : valorsifalso`).

Sabiendo esto, y que la última expresión de una función Lisp es el valor devuelto por ésta, se puede entender el segundo ejemplo mucho mejor.

Pensar en el objetivo, no en el proceso

Aunque el ejemplo de arriba es bastante simplista, también ilustra que los lenguajes funcionales tienden a hacerte pensar más en el objetivo que en el proceso. Para algoritmos, generalmente una solución en un lenguaje funcional se parece más a la definición matemática. En el primer ejemplo, la implementación de la función está basada en el proceso de calcular. En el segundo, en el significado de las operaciones.

Si lo pensamos en castellano, el primer ejemplo sería algo así:

> Primero multiplicamos `cost` por `nitems` para calcular el precio base, y lo guardamos en `total-price`. Si éste está por encima de `limit`, entonces primero calculamos `tax` a base de multiplicar `total-price` por `percent-tax` dividido por 100, y luego guardamos en `total-price` la suma del antiguo `total-price` más `tax`. El resultado es el valor guardado en `total-price` al final del proceso.

El segundo ejemplo, en cambio, sería algo así:

> El resultado es la suma de `total-price` y `tax` si `total-price` está por encima de `limit`, o `total-price` en caso contrario. Definimos `total-price` como la multiplicación de `cost` por `nitems`, y `tax` como la multiplicación de `total-price` por `percent-tax` dividido por 100.

Nótese cómo en la segunda explicación podemos dejar para el final la explicación de los valores declarados en el bloque `let*`. En muchos casos, y siempre que hayamos escogido buenos nombres para esos valores, no hará falta leer el bloque `let*` para entender la función.

Usar variables que no cambian una vez les hemos asignado un valor, es una buena costumbre porque hace más fácil entender de dónde sale cada valor. Por esa razón, el lenguaje Scala distingue entre dos tipos de variables: `var` y `val`. El segundo tipo, que es el más usado con diferencia, declara una variable *inmutable*, por lo que el compilador nos asegura que no podemos asignar ningún

otro valor una vez declarada.

Esto está relacionado con el siguiente apartado, «Diseño de abajo a arriba, funciones pequeñas»: para poder pensar en el significado, muchas de las operaciones tienen que abstraerse en pequeñas funciones.

Diseño de abajo a arriba, funciones pequeñas

Otra característica común de la programación en Lisp es intentar acercar el lenguaje a la tarea que se intenta resolver. Una de las maneras de hacerlo es escribir funciones que, aunque sean pequeñas y simples, escondan detalles de implementación que no nos interesen y que suban el nivel de abstracción. La lección aquí es que escribir funciones y métodos de una o dos líneas es útil *si* suben el nivel de abstracción. No se trata de «esconder código» para no tenerlo a la vista, se trata de hacer olvidar a uno mismo parte de la complejidad de la tarea que está realizando.

Como ejemplo, sigamos con el caso anterior de los precios y los artículos. Una de las operaciones que hacemos es calcular un tanto por ciento. Si suponemos que es una operación que usaremos más de una vez, podría tener sentido abstraer ese cálculo. No nos vamos a ahorrar líneas de código, pero esta versión puede requerir menos esfuerzo mental y ser más fácil de leer (imagínese el siguiente ejemplo en castellano, tal y como hicimos en el anterior apartado):

```
; En el caso de Lisp, podríamos haber llamado a esta función '%', de modo que
; la llamada de abajo quedaría '(% percent-tax total-price)'
(defun percentage (percent amount)
  (* amount (/ percent 100)))

(defun calculate-price (cost nitems limit percent-tax)
  (let* ((total-price (* cost nitems))
         (tax         (percentage percent-tax total-price)))
    (if (> total-price limit)
      (+ total-price tax)
      total-price)))
```

Solución abstrayendo el cálculo de porcentajes (calculate-prices.lisp)

Calcular un tanto por ciento es trivial, y por escribir la función `percentage` no estamos ahorrando líneas de código, pero cada segundo que ahorramos en entender trivialidades al leer la fuente es un segundo más que podemos dedicar a asuntos más importantes. Y el tiempo que necesitamos para entender código sin las abstracciones apropiadas, con frecuencia crece exponencialmente, no linealmente, al añadir nuevas faltas de abstracción.

Otra ventaja de abstraer funciones de esta manera es que estas funciones normalmente son bastante fáciles de probar, porque tienden a tener interfaces sencillas y responsabilidades claras. En el caso de lenguajes que tienen una consola interactiva (como Lisp, Python, Ruby y otros) es fácil experimentar con la función y ver lo que hace, facilitando la escritura de pruebas unitarias en cualquier lenguaje. Especialmente si evitamos los efectos colaterales, como veremos en el siguiente apartado.

Efectos colaterales

Los llamados *efectos colaterales* son uno de los conceptos más importantes de la programación funcional, por no decir que el más importante. Es lo que diferencia los lenguajes puramente funcionales de los lenguajes funcionales no puros. Incluso los programadores de los lenguajes que no son puramente funcionales (como Lisp) generalmente intentan evitar efectos colaterales.

Un efecto colateral es cualquier cambio que una función produce fuera del ámbito de la función en sí. Por ejemplo, una función que modifique una variable que ha recibido como parámetro (es decir, «parámetros de entrada/salida») o que modifique variables globales o cualquier otra cosa que no sean variables locales a la función está produciendo efectos colaterales. Esto incluye cualquier tipo de entrada/salida, como leer o escribir ficheros o interactuar con la pantalla, el teclado o el ratón.

¿Por qué es tan importante evitar efectos colaterales? De nuevo, como en el caso de las pequeñas funciones que suban el nivel de abstracción, evitar un solo efecto colateral no es una ventaja muy grande. Sin embargo, evitar efectos colaterales como regla general hace que los programas sean más fáciles de entender y mantener, y que haya menos sorpresas. La razón es que evitar efectos colaterales *garantiza* que ningún error en la función pueda afectar a nada más. Si además no hacemos referencia a nada externo a la función, como variables globales, tenemos una garantía extra importantísima: la función es independiente del resto del código, lo que significa que ningún fallo del resto del programa puede afectar a nuestra función, y que podemos probar la función independientemente del resto del código, lo cual no sólo es práctico, sino que hace más fácil asegurarse de que cubrimos todos los casos posibles de la función con baterías de pruebas.

Veamos un ejemplo de efectos colaterales en Python. El método `sort`, desgraciadamente, modifica la lista sobre la que se llama. Esto puede producir sorpresas desagradables, como veremos en el primer ejemplo. Digamos que estamos escribiendo un programa para gestionar competiciones de carreras y escribimos una función `best_time` que recibe una lista de números y devuelve el menor (obviamos la existencia de la función `min` para hacer el ejemplo más ilustrativo):

```
def best_time_BAD(list):
  if len(list) == 0:
    return None
  list.sort()
  return list[0]

times = [5, 9, 4, 6, 10, 8]
best_time_BAD(times)   # Devuelve 4
print times            # ¡Esto imprime «[4, 5, 6, 8, 9, 10]»!
```

Sorpresa desagradable debida a un efecto colateral (best-time-bad.py)

Una forma de resolver este problema es usar la función `sorted` en vez del método `sort`:

```
def best_time(list):
  if len(list) == 0:
    return None
  return sorted(list)[0]
```

```
times = [5, 9, 4, 6, 10, 8]
best_time(times)   # Devuelve 4
print times        # Imprime «[5, 9, 4, 6, 10, 8]»
```

Mejor implementación, sin efectos colaterales (best-time-bad.py)

En Ruby normalmente se usa la convención de añadir un «!» al final del nombre del método si éste produce efectos colaterales (otra convención que se puede apreciar en el ejemplo es cómo los métodos que devuelven verdadero/falso terminan en «?»). El ejemplo de arriba se podría traducir a Ruby de la siguiente manera:

```
require 'pp'              # Pretty printer

def best_time_BAD(list)
  if list.empty?
    nil
  else
    list.sort!          # «sort!», ¡con efectos colaterales!
    list[0]
  end
end

times = [5, 9, 4, 6, 10, 8]
best_time_BAD(times)   # Devuelve 4
pp times               # Imprime «[4, 5, 6, 8, 9, 10]»

def best_time(list)
  if list.empty?
    nil
  else
    list.sort[0]        # «sort», sin «!»
  end
end

times2 = [5, 9, 4, 6, 10, 8]
best_time(times2)   # Devuelve 4
pp times2           # Imprime «[5, 9, 4, 6, 10, 8]»
```

Efectos colaterales en Ruby (best-time.rb)

Por último, evitar efectos colaterales permite a las funciones usar una técnica de optimización llamada «memorización» (*memoization* en inglés). Esta técnica consiste en recordar el valor retornado por la función la primera vez que se llama. Cuando se vuelve a llamar a la función con los mismos parámetros, en vez de ejecutar el cuerpo de la función, se devuelve el valor recordado. Si la función no produce ningún efecto colateral, esta técnica es perfectamente segura porque está garantizado que los mismos parámetros de entrada siempre producen el mismo resultado. Un ejemplo muy sencillo de memorización en Javascript es la siguiente implementación de la serie de Fibonacci:


```
var fibonacciCache = {0: 1, 1: 1};

function fibonacci(pos) {
  if (pos < 0) {
    throw "La serie de Fibonacci sólo está definida para números naturales";
  }

  if (! fibonacciCache.hasOwnProperty(pos)) {
    console.log("Calculo el resultado para la posición " + pos);
    fibonacciCache[pos] = fibonacci(pos - 1) + fibonacci(pos - 2);
  }

  return fibonacciCache[pos];
}
```

Implementación de la serie de Fibonacci con memorización (fibonacci.js)

Si se copia este código en una consola Javascript (digamos, Node) y se hacen distintas llamadas a la función `fibonacci`, se podrá comprobar (gracias a los mensajes impresos por `console.log`) que cada posición de la serie sólo se calcula una vez.

En lenguajes dinámicos como Python, Ruby o Javascript, es relativamente sencillo escribir una función que reciba otra función como parámetro y le aplique la técnica de «memorización». El siguiente apartado explora la técnica de manipular funciones como datos.

Funciones de orden superior

Otra de las características comunes de los lenguajes funcionales es tratar a las funciones como «ciudadanos de primera clase». Es decir, las funciones son valores más o menos normales que se pueden pasar como parámetros, asignar a variables y devolver como resultado de la llamada a una función. Las funciones que utilizan esta característica, es decir, que manipulan o devuelven funciones, reciben el nombre de *funciones de orden superior*. Afortunadamente, muchos lenguajes populares tienen este tipo de funciones.

La primera vez que uno se encuentra funciones de orden superior puede pensar que sus usos son limitados, pero realmente tienen muchas aplicaciones. Por un lado, tenemos las funciones y métodos que traiga el lenguaje de serie, por lo general de manejo de listas. Por otro, tenemos la posibilidad de escribir nuestras propias funciones y métodos de orden superior, para separar o reutilizar código de manera más efectiva.

Veamos un ejemplo de lo primero en Ruby. Algunos de los métodos de la clase `Array` reciben una función como parámetro (en Ruby se los llama *bloques*), lo que permite escribir código bastante compacto y expresivo:

```
# Comprobar si todas las palabras tienen menos de 5 letras
if words.all? {|w| w.length < 5 }
```

```ruby
    # ...
end

# Comprobar si el cliente tiene algún envío pendiente
if customer.orders.any? {|o| not o.sent? }
    # ...
end

# Obtener las asignaturas suspendidas por un alumno
failed_subjects = student.subjects.find_all {|s| s.mark < 5 }

# Dividir los candidatos entre los que saben más de
# dos idiomas y los demás
polyglots, rest = cands.partition {|c| c.languages.length > 2 }

# Obtener una versión en mayúsculas de las palabras
# de la lista
words = ["hoygan", "kiero", "hanime", "gratix"]
shouts = words.map {|w| w.upcase}
```

Métodos de orden superior en Ruby

El código equivalente que habría que escribir para conseguir el mismo resultado sin funciones de orden superior es bastante más largo y difícil de leer. Además, si quisiéramos hacer operaciones parecidas variando la condición (digamos, en una parte del código queremos comprobar si todas las palabras tienen menos de cinco letras, y en otra queremos comprobar si todas las palabras se componen exclusivamente de letras, sin números u otros caracteres) el código empeoraría rápidamente.

Escribir nuestras propias funciones tampoco tiene que ser difícil, ni usarse en casos muy especiales. Pueden ser usos tan comunes y sencillos como el siguiente ejemplo en Javascript:

```javascript
// Queremos poder escribir el siguiente código
var comicCollection = new Database('comics');
comicCollection.onAdd(function(comic) {
    console.log("Nuevo cómic añadido: " + comic.title);
});
// La siguiente línea debería imprimir «Nuevo cómic...» en la consola
comicCollection.add({title:  "Batman: The Dark Knight Returns",
                     author: "Frank Miller"});

// La implementación de onAdd puede ser muy sencilla
Database.prototype.onAdd = function(f) {
    this.onAddFunction = f;
}
// La implementación de add también
Database.prototype.add = function(obj) {
    this.data.push(obj);
    if (typeof(this.onAddFunction) === 'function') {
        this.onAddFunction(obj);
    }
}
```


Funciones de orden superior en Javascript

A partir de Ecmascript 5, la clase `Array` añade varios métodos de orden superior que son comunes en la programación funcional.

Evaluación perezosa

La última característica de lenguajes funcionales que exploraremos es la *evaluación perezosa*. No hay muchos lenguajes que incluyan evaluación perezosa, pero se puede imitar hasta cierto punto, y saber cómo funciona puede darnos ideas e inspirarnos a la hora de diseñar nuestros propios sistemas. Uno de los relativamente pocos lenguajes que incluye evaluación perezosa es Haskell.

La evaluación perezosa consiste en no hacer cálculos que no sean necesarios. Por ejemplo, digamos que escribimos una función que genere recursivamente una lista de 10 elementos, y otra función que llame a la primera, pero que sólo use el valor del cuarto elemento. Cuando se ejecute la segunda función, Haskell ejecutará la primera hasta que el cuarto elemento sea calculado. Es decir: Haskell no ejecutará, como la mayoría de los lenguajes, la primera función hasta que *devuelva* su valor (una lista de 10 elementos); sólo ejecutará la función hasta que se *genere* el cuarto elemento de la lista, que es lo único que necesita para continuar la ejecución del programa principal. En este sentido, la primera función es como una expresión matemática: inicialmente Haskell no conoce el valor de la expresión, y sólo calculará la parte de ésta que necesite. En este caso, los cuatro primeros elementos.

¿Cuál es la ventaja de la evaluación perezosa? En la mayoría de los casos, eficiencia. En otros casos, legibilidad. Cuando no tenemos que preocuparnos por la memoria o ciclos de CPU usados por nuestra función, podemos hacer que devuelvan (teóricamente) listas o estructuras infinitas, las cuales pueden ser más fáciles de leer o implementar en algunos casos. Aunque no es el ejemplo más claro de legibilidad de evaluación perezosa, entender la siguiente implementación de la serie de Fibonacci, aclarará la diferencia con la evaluación estricta. Nótese que la función calcula la serie *entera*, es decir, una lista *infinita*:

```haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Implementación de la serie de Fibonacci, en Haskell

Normalmente la función es imposible de entender de un primer vistazo si no estamos familiarizados con la programación funcional y la evaluación perezosa, pero hay varios puntos que nos ayudarán:

1. `tail lista` devuelve la lista dada, saltándose el primer elemento. Es decir, si `lista` es `(1 2 3)`, `tail lista` es `(2 3)`.
2. `zipWith` calcula, dada una operación y dos listas, una lista que tiene: como primer elemento, el resultado de aplicar la operación dada al primer elemento de las dos listas; como segundo el resultado de aplicar la operación al segundo elemento de las dos listas; etc. Así, `zipWith` llamado con la función suma y las listas `(1 2 3)` y `(0 1 5)` resultaría en `(1 3 8)`.

3. Cada elemento de la lista devuelta por `fibs` se calculará individualmente, y estará disponible en memoria sin necesidad de volver a ejecutar el código de la función.

Así, lo que ocurre es:

1. Haskell empieza a construir una lista con los elementos `0` y `1`. En este punto, `fibs = (0 1)`.
2. El tercer elemento será el primer elemento de la subexpresión `zipWith ...`. Para calcularlo, necesitamos la lista `fibs` (por ahora `(0 1)`, ya que sólo conocemos dos elementos) y `tail fibs` (por ahora `(1)`). Al sumar el primer elemento de cada una de esas listas (`0` y `1`), el resultado es `1`. En este punto, `fibs = (0 1 1)` y la subexpresión `zipWith ... = (1)`.
3. El cuarto elemento de `fibs` es el segundo elemento de `zipWidth ...`. Para calcularlo necesitaremos el segundo elemento de `fibs` y el segundo elemento de `tail fibs`. El segundo elemento de `tail fibs` es el tercer elemento de `fibs`, que ya conocemos porque lo calculamos en el paso anterior. Nótese que *no* hace falta ninguna llamada recursiva porque los valores que necesitamos ya están calculados. La evaluación perezosa funciona como una función matemática: no hace falta que volvamos a calcular un valor si ya sabemos el resultado. En este punto, `fibs = (0 1 1 2)` y la subexpresión `zipWith ... = (1 2)`.
4. Para el quinto elemento (el tercero de `zipWidth`), necesitaremos el tercer y cuarto elementos de `fibs`, que llegados a este punto ya conocemos porque los hemos calculado en los pasos anteriores. Y así sucesivamente.

Estos pasos no se ejecutan indefinidamente: se irán ejecutando hasta que se obtenga el elemento de `fibs` que se necesite. Es decir, si asignamos `fibs` a una variable pero nunca la usamos, el código no se ejecutará en absoluto; si usamos el valor del tercer elemento de la serie en algún cálculo, sólo se ejecutarán los dos primeros pasos descritos arriba; etc. En ningún caso se intenta ejecutar `fibs` hasta que devuelva «el valor completo».

La evaluación perezosa se puede ver como aplicar la técnica de «memorización» automáticamente a todo el lenguaje. Un posible uso es calcular tablas de valores que son lentos de calcular: en algunos casos podríamos cargar una tabla precalculada en memoria, pero el coste puede ser prohibitivo si la tabla es grande o potencialmente infinita.

Conclusión

Aprender lenguajes nuevos, especialmente de paradigmas con los que estamos menos familiarizados, nos puede enseñar muchas cosas sobre programación en general. Este proceso de aprendizaje nos hará mejores programadores, y muchas de esas lecciones serán aplicables a todos los lenguajes que conozcamos, no sólo a los lenguajes similares al que acabemos de aprender. En particular, los lenguajes funcionales son suficientemente accessibles y similares a los lenguajes más populares como para enseñarnos muchas lecciones útiles.

Documentación activa

Joaquín Caraballo

Si en un extremo están los que definen el problema a resolver y en el otro los que lo resuelven, ¿cómo podemos asegurarnos de que todos estamos intentando resolver el mismo problema? Y, si aceptamos que la definición va a necesitar ser enriquecida, corregida, matizada, e incluso que el problema a resolver va a cambiar radicalmente durante la vida del proyecto, ¿cómo mantenemos una misma definición para todos los interesados, y más importante, cómo aseguramos que nuestro programa resuelve el problema?

Las pruebas funcionales

Para tener una cierta confianza en que la aplicación resuelve un cierto problema se llevan a cabo pruebas funcionales; estas resuelven un ejemplo concreto y representativo simulando las acciones del usuario y verificando que la reacción de la aplicación es la esperada.

Las pruebas funcionales han de mantenerse al día y evolucionar con los cambios de la aplicación y sus requisitos. Otro de los retos es que sean correctas, es decir, que el comportamiento que estén verificando sea realmente el que se requiere de la aplicación.

Es crucial que las pruebas funcionales estén automatizadas; lo típico es escribirlas en el mismo lenguaje de programación de la aplicación. Esto nos permite ejecutar un gran número de ellas de forma sistemática y tras cada cambio, con lo que también nos protegerán de que algo que funcionaba deje de hacerlo, es decir, de posibles *regresiones*.

neverread

Supongamos que tenemos un cliente con falta de tiempo para leer artículos de Internet.

> -Tantos artículos interesantes, no tengo tiempo para leerlos, pero tampoco puedo ignorarlos. Me gustaría tener una aplicación en la que copiar la dirección del artículo que me interese.
>
> -¡Ah!, ¿quieres una aplicación que te permita almacenar enlaces a artículos? La aplicación mantendría una lista con los artículos almacenados, a la que luego podrías acceder cuando tengas tiempo y leer los artículos...
>
> -No, no, si yo lo que quiero es que los enlaces a artículos desaparezcan, si tuviera tiempo para leerlos después usaría instapaper, hombre. Estaría bien que tuviese una lista de artículos por leer siempre vacía, me daría una sensación genial de tenerlo todo bajo control.
>
> -Er... vale.

De la conversación hemos conseguido extraer un criterio de aceptación:

> Por mucho que añadamos artículos, estos no se añadirán a una lista de artículos por leer.

La prueba funcional correspondiente, verificará que la aplicación cumple el criterio para un ejemplo concreto:

> Cuando añadimos el artículo `art.culo/interesante.html`, la lista de artículos por leer permanece vacía.

Las pruebas funcionales, si bien no tienen necesariamente que serlo, se llevan a cabo con frecuencia como pruebas de punta a punta *(end to end)* ([goos] pág 10), es decir, pruebas en las que se ejercita la aplicación en su conjunto y desde afuera, simulando las acciones del usuario y de los sistemas colaboradores, y evaluando la corrección según la perciben usuario y colaboradores.

Hemos decidido resolver el problema propuesto con una aplicación web, *neverread*. Implementaremos en Java una prueba funcional de punta a punta que verifique el criterio de aceptación con una prueba *junit* que va a iniciar la aplicación y a continuación ejercitarla y evaluarla. Para simular la interacción de un usuario a través de un navegador web utilizaremos Selenium/WebDriver/HtmlUnit:

```
public class ListStaysEmptyTest {
    private WebDriver webDriver;
    private NeverReadServer neverread;

    @Test
    public void articleListStaysEmptyWhenAddingNewArticle() {
        webDriver.findElement(By.name("url")).sendKeys("interesting/article.html", Keys.ENTER);
        assertThat(webDriver.findElements(By.cssSelector("li")).size(), is(0));
    }

    @Before
    public void setUp() throws Exception {
        neverread = new NeverReadServer();
        neverread.start(8081);

        WebDriver driver = new HtmlUnitDriver();
        driver.get("http://localhost:8081");
        webDriver = driver;
    }

    @After
    public void tearDown() throws Exception {
        webDriver.close();
        neverread.stop();
    }
}
```

Prueba punta a punta en Java (purejava/ListStaysEmptyTest.java)

Documentación activa

Si bien podríamos considerar que nuestra prueba funcional en Java es la documentación principal donde se registran los criterios de aceptación; dependiendo de quién vaya a consumirla, esto puede o no ser una opción. Además, conseguir reemplazar la legibilidad de un párrafo en español con una prueba en el lenguaje de programación es todo un reto, particularmente si el lenguaje de programación es tan prolijo como Java.

En consecuencia, en la mayoría de los proyectos es necesario documentar los requisitos de manera más textual.

Una forma de intentar obtener lo mejor de las dos alternativas es conectar la documentación a la aplicación de tal forma que en todo momento sea automático y evidente verificar el cumplimiento de cada uno de los requisitos expresados, lo que nos ayudará a mantener la documentación sincronizada con la aplicación y con el cliente.

La documentación, junto con la capacidad de procesarla para su verificación, servirá así de batería de pruebas funcionales; si ejecutamos esta batería con frecuencia (idealmente con cada cambio) y la mantenemos veraz, estaremos garantizando el correcto funcionamiento de la aplicación... para la definición especificada.

Concordion

Concordion es una de las herramientas que nos pueden ayudar a conectar la documentación con la aplicación; en Concordion los criterios se escriben en HTML al que se añaden algunas marcas que comienzan con `concordion:`

```
<p>
  When an article is added, the list of unread articles stays empty
</p>

<div class="example">
<h3>Example</h3>

<p>
  When the article
  <b concordion:set="#url">interesting/article.html</b>
  is added, the list of unread articles stays
  <b concordion:assertEquals="articleListAfterAdding(#url)">empty</b>
</p>
</div>
```

Primera prueba usando Concordion (concordion/v1/ListStaysEmpty.html)

Vemos que hemos expresado el ejemplo en forma de condición y consecuencia que se debe verificar. Los atributos `concordion:set` y `concordion:assertEquals` conectan el documento al método de la clase de respaldo, escrita en Java, `String articleListAfterAdding(String url)`, que se encargará de hacer lo que dice el texto.

```
public class ListStaysEmptyTest extends ConcordionTestCase {
    private WebDriver webDriver;
    private NeverReadServer neverread;
```

```
    @SuppressWarnings(value = "unused")
    public String articleListAfterAdding(String url) throws InterruptedException {
        webDriver.findElement(By.name("url")).sendKeys(url, Keys.ENTER);
        List<WebElement> pendingArticles = webDriver.findElements(By.cssSelector("li"));

        return convertListOfArticlesToString(pendingArticles);
    }

    private static String convertListOfArticlesToString(List<WebElement> pendingArticles) {
        if (pendingArticles.isEmpty()) return "empty";
        else {
            StringBuilder stringBuilder = new StringBuilder();
            stringBuilder.append(pendingArticles.get(0).getText());

            for (int i = 1; i < pendingArticles.size(); i++)
                stringBuilder.append(", ").append(pendingArticles.get(i).getText());

            return stringBuilder.toString();
        }
    }
```

Clase de respaldo de la prueba en Concordion (purejava/ListStaysEmptyTest.java)

Al ejecutar `ListaPermaneceVaciaTest`, Concordion generará un documento HTML con el texto anterior en el que se indicará si se cumple la aserción resaltándola en verde o rojo.

Paso a paso

Veamos qué es lo que está pasando aquí. Hemos escrito el criterio de aceptación en `ListaPermaneceVacia.html`. Acompañando al HTML hemos escrito una clase en Java que extiende una clase de infrastructura de Concordion: `class ListaPermaneceVaciaTest extends ConcordionTestCase`.

Cuando ejecutamos `ListaPermaneceVaciaTest`:

1. Concordion procesa el HTML.
2. Concordion detecta la marca `concordion:set="#url"` y guarda el contenido de esa marca HTML (en este caso, «art.culo/interesante.html») en la variable de Concordion `#url`.
3. Concordion detecta la marca `concordion:assertEquals="articleListAfterAdding(#url)"`, por lo que busca en la clase de acompañamiento un método denominado `articleListAfterAdding` y lo ejecuta, pasándole el contenido de `#url` como parámetro.
4. El método `articleListAfterAdding` simula la acción de un usuario que introduce `url` y obtiene la lista de artículos resultante.
5. Mediante `convertListOfArticlesToString`, transformamos la lista producida por WebDriver en una representación textual que pueda ser comparada con el texto del HTML. Hemos decidido que la representación textual de una lista vacía sea «vacía».
6. El método `articleListAfterAdding` retorna, devolviendo una cadena (en este caso «vacía») que es comparada con el contenido de la marca HTML en el que se encontró `concordion:assertEquals`.
7. Concordion termina de procesar el documento HTML y genera otro HTML en el que el texto que tiene la marca `concordion:assertEquals` está resaltado en verde, para indicar que la aserción se cumple.

Manteniendo el nivel de abstracción apropiado

Es importante esforzarse en describir el funcionamiento de la aplicación en términos del dominio. Por ejemplo, podríamos haber caído en la tentación de escribir el ejemplo como *Cuando el usuario entra una cadena en la caja de texto y pulsa enter, la lista de artículos está vacía*. Sin embargo, eso sería perjudicial porque nos alejaría de la persona que define lo que debe hacer la aplicación y resultaría más *frágil*, es decir, en cuanto decidiéramos cambiar la implementación, por ejemplo, supongamos que las direcciones se introducen arrastrándolas a una zona de la aplicación, tendríamos que reescribir el documento.

A poco que la documentación activa crezca, las clases de respaldo van a necesitar una cantidad importante de código. Algunas abstracciones pueden ayudarnos reducir la repetición y la fragilidad de las clases de respaldo.

Podemos hacer que la clase de respaldo sólo *hable* en el lenguaje del dominio, para lo cual hemos de desarrollar un lenguaje dedicado, con lo que el método quedaría algo así:

```
public String articleListAfterAdding(String article) throws InterruptedException {
    driver.addArticle(article);
    return convertListOfArticlesToString(driver.getListOfArticles());
}
```

Método de respaldo usando lenguaje del dominio (concordion/v2_appdriver/ListStaysEmptyTest.java)

Otra posibilidad es abstraer la página web en términos de los elementos del entorno gráfico, es decir, que hable de elementos visuales de la página.

```
public String articleListAfterAdding(String url) throws InterruptedException {
    page.enterIntoNewArticlesTextBox(url);
    List<String> pendingArticles = page.getArticlesInListOfArticles();

    return convertListOfArticlesToString(pendingArticles);
}
```

Método de respaldo usando lenguaje del entorno gráfico (concordion/v3_pagedriver/ListaStaysEmptyTest.java)

La capa de abstracción en términos de lenguaje del dominio es la opción más *pura*, pero dependiendo del proyecto podremos

preferir una capa que se exprese en términos gráficos o ambas, dependiendo de la complejidad del proyecto y de cuán involucrado esté el cliente en los detalles gráficos.

Pruebas asíncronas

En las secciones anteriores nos hemos permitido hacer un poco *trampas* que deberíamos descubrir antes de cerrar el artículo. Supongamos que el desarrollador, al escribir el código de la aplicación comete una equivocación por no entender debidamente lo que necesita el cliente; decide hacer una aplicación que *añade* los artículos a una lista de artículos a leer. Nuestras pruebas funcionales deberían detectar este error marcando la aserción en rojo. Sin embargo, nos encontramos con que la pruebas pasan.

Evidentemente, nuestras pruebas funcionales no son correctas, esto se debe a que estamos verificando que el estado de la lista de artículos es el mismo después de entrar el nuevo artículo que antes de entrarlo, y la prueba verifica la condición antes de que la aplicación tenga tiempo de añadir erróneamente artículos a la lista.

Probar sistemas asíncronos es lo suficientemente complejo como para justificar un artículo en sí mismo, pero si enumeramos algunas de las opciones, de más rápidas y sencillas de implementar a menos, tenemos:

1. Probamos sólo la parte síncrona del sistema. Esto hace las pruebas más sencillas y rápidas a costa de reducir el alcance.
2. Introducimos puntos de sincronización. Volveremos a este en un segundo.
3. Verificamos periódicamente la aserción hasta que se cumpla o se agote el tiempo de espera. En esta opción es crucial ajustar la duración, si esperamos demasiado las pruebas tardarán demasiado innecesariamente, si esperamos demasiado poco tendremos falsos negativos.

En nuestro ejemplo sabemos que, cada vez que la aplicación responde a la entrada de un nuevo artículo, lo último que hace es borrar la caja de texto. Por lo tanto, podemos utilizar este evento como punto de sincronización, es decir, antes de verificar que la lista permanece vacía esperaremos a que la caja se haya borrado.

```
public void addArticle(String url) {
    webDriver.findElement(By.name("url")).sendKeys(url, Keys.ENTER);

    new WebDriverWait(webDriver, 2).until(new ExpectedCondition<Object>() {
        @Override
        public Object apply(WebDriver webDriver) {
            return "".equals(webDriver.findElement(By.name("url")).getAttribute("value"));
        }
    });
}

public List<String> getListOfArticles() {
    return webElementsToTheirTexts(webDriver.findElements(By.cssSelector("li")));
}
```

Ejemplo de punto de sincronización (concordion/v5_with_synchronisation/tools/NeverReadDriver.java)

Conclusión

La *documentación activa* es una forma de probar funcionalmente un programa en el que cada criterio de aceptación se enuncia con un texto que se enlaza a la ejecución de código que verifica el criterio. Al ejecutarla, se produce un resultado que indica, de forma legible para los expertos del dominio, qué criterios de aceptación cumple el programa y qué criterios no cumple. Como casi todo, su uso debería adaptarse a la composición del equipo y la complejidad del proyecto.

Siete problemas al probar programas

Esteban Manchado Velázquez

La mayoría de los profesionales de la informática coincidirán en que probar es una de las tareas fundamentales del desarrollo, pero si ya es difícil aprender técnicas de programación, mucho más difícil es aprender técnicas de pruebas, tanto manuales como automáticas. Primero, porque desgraciadamente es un conocimiento menos extendido. Segundo, porque es aún más abstracto que la programación.

Por ello, todos los consejos y la experiencia compartida de los que nos podamos aprovechar son especialmente importantes. Los siguientes problemas están centrados en las pruebas automáticas, pero muchos de ellos ocurren también al hacer pruebas manuales. Están ordenados por experiencia: el primer problema aparece principalmente en equipos que nunca han escrito pruebas, y el último lo tienen incluso desarrolladores con años de experiencia.

Como todo, hay que aplicar las soluciones propuestas entendiendo por qué, cómo y cuándo son aplicables y útiles, nunca siguiéndolas ciegamente en todas las situaciones.

Dejar las pruebas para el final

Aunque es normal que al acercarse el final de cada ciclo de desarrollo se intensifique el esfuerzo de pruebas (especialmente las manuales), es un error mayúsculo no haber probado desde el principio del desarrollo. Esto no es un tópico o una consideración teórica o académica: no probar desde el principio del ciclo de desarrollo tiene muchas desventajas. Por ejemplo:

1. El esfuerzo mental de probar un programa grande es mucho mayor que el esfuerzo de probar un programa pequeño. No se sabe por dónde empezar, es difícil saber cuándo terminar, y es fácil quedarse con la sensación de que no todo está probado correctamente. Si probamos los componentes que vamos creando desde el principio, es mucho más fácil saber cómo atacar el problema y hacer pruebas más completas.
2. Si probamos mientras escribimos código y sabemos el nivel de calidad de éste, será más fácil hacer estimaciones sobre los recursos necesarios para terminar el resto del trabajo. Esto nos dará mucha más flexibilidad para renegociar fechas o las

características que se tendrán que quedar fuera, en vez de pasar al estado de alarma cuando nos demos cuenta, demasiado tarde, de que no tenemos tiempo de arreglar los fallos que encontremos los últimos días.

3. Cuando hemos «terminado» un proyecto y sólo queda probar, es normal tener la tendencia a «no querer ver fallos» o restarles importancia, a que nos inunde un falso optimismo que confirme que de verdad hemos terminado y no queda nada por hacer. Sobre todo si quedan pocos días hasta la entrega y sentimos que no podemos hacer gran cosa, y sólo queremos entregar el resultado y olvidarnos del proyecto. Esto ocurre mucho menos si tenemos a profesionales exclusivamente dedicados a las pruebas, claro.
4. Probar desde el principio significa que estaremos probando durante más tiempo, por lo que habremos encontrado más casos que puedan ser problemáticos, y por tanto más fallos (y las soluciones a éstos), antes de que sea demasiado tarde.
5. Todas las pruebas que hagamos pronto ayudarán a aumentar la calidad del código, lo que no sólo afecta a la calidad final del producto, sino a lo fácil que es escribir y depurar cualquier otro código que dependa del primero. Los fallos de interacción entre las dos bases de código serán más fáciles de encontrar y, en particular, la fuente de los errores.
6. Si desde el principio escribimos pruebas automáticas, nos obligaremos a nosotros mismos a escribir APIs más limpias. Generalmente, código más fácil de probar es código más desacoplado, más limpio y más estable. Indudablemente, una de las ventajas de escribir pruebas automáticas es que ayudan a mantener un diseño de mayor calidad. Pero si no probamos desde el principio, perdemos esta ventaja.
7. Al no probar desde el principio, desarrollaremos código que no es especialmente fácil de probar, y cuanto más tarde empecemos a adaptar el código para que lo sea, más esfuerzo costará. Una vez hayamos escrito una gran cantidad de código sin tener en cuenta si es fácil de probar o no, la tentación de dejar muchas partes sin pruebas será irresistible. Y, por supuesto, si no hacemos el esfuerzo de adaptar el código en ese momento, entraremos en un círculo vicioso.

Como tarde o temprano se tendrá que probar el resultado del trabajo, mejor empezar temprano porque es menos costoso (en tiempo y esfuerzo mental) y los resultados son mejores. Escribir código sin probar es simplemente irresponsable y una falta de respeto con respecto a los usuarios del producto *y* al resto de los miembros del equipo, especialmente los que tengan que mantener el código después y cualquier miembro del equipo que use directa o indirectamente el código sin probar.

Ser demasiado específico en las comprobaciones

Esto es un problema bastante común, sobre todo cuando se empiezan a hacer pruebas. Las pruebas automáticas son, de alguna manera, una descripción de lo que se espera que el programa haga. Una especificación en código, por así decirlo. Como tal, sólo debe describir el comportamiento que esperamos que no cambie. Si somos demasiado específicos o exigentes en las comprobaciones de nuestras pruebas, éstas no sólo evitarán que introduzcamos fallos en el código, sino también que podamos hacer otro tipo de cambios.

Por ejemplo, digamos que estamos escribiendo una aplicación de tareas pendientes muy sencilla. La clase que representa una lista de tareas se llama BrownList, y podría tener una pinta así en Ruby:

```ruby
class BrownList
  def initialize(db)
    # ...
  end

  def add_brown(title)
    # Insertamos en una base de datos, devolvemos un id
  end

  def mark_brown_done(id)
    # Marcamos el id dado como hecho
  end

  def pending_browns
    # Devolvemos una lista de las tareas pendientes
  end
end

# Se usa así
bl = BrownList.new(db_handle)
id = bl.add_brown("Tarea pendiente")
bl.mark_brown_done(id)
lista_tareas_pendientes = bl.pending_browns
```


Ejemplo de implementación de la clase `BrownList`

Ahora, para probar que el método `add_brown` funciona, puede que se nos ocurra conectarnos a la base de datos y comprobar si tiene la fila correcta. En la gran mayoría de los casos esto es un error. Para entender por qué, hay que darse cuenta de que las pruebas definen *qué* significa que el código funcione. Así, si las pruebas usan detalles de la implementación, se estará definiendo de manera implícita que el programa sólo «funciona bien» si mantiene los mismos detalles de implementación. Lo cual es, naturalmente, un error, porque no permite que el código evolucione.

```ruby
class TestBrownList < Test::Unit::TestCase
  def setup
    # Recreamos la base de datos de pruebas para que esté vacía
  end

  def test_add_brown_simple_____MAL
    db_handle = ...
    bl = BrownList.new(db_handle)
    bl.add_brown("Tarea de prueba")

    count = db_handle.execute("SELECT COUNT(*) FROM browns")
    assert_equal 1, count
  end
```

```
end
```

Mal ejemplo de prueba del método `add_brown` de `BrownList`

En este ejemplo concreto, hay muchos casos en los que esta prueba fallaría, a pesar de que el código podría estar funcionando perfectamente:

- El nombre de la tabla donde guardamos las tareas cambia
- Añadimos la característica multiusuario, por lo que podría haber más tareas en esa tabla de las que queremos contar
- Decidimos que vamos a usar una base de datos no-SQL, por lo que la manera de contar cambiaría
- Añadimos un paso intermedio de algún tipo, de tal manera que las tareas no se crearían inicialmente en la base de datos, sino en algo como *memcached*, y unos segundos después irían a la base de datos

Las pruebas no deben limitarnos cuando reorganizamos código o cambiamos detalles de implementación. De hecho, una de las ventajas de tener pruebas automáticas es que cuando reorganicemos código, sabremos si estamos haciendo algo mal porque las pruebas fallarán. Si no estamos seguros de que cuando una prueba falla es porque hay un problema en el código, nuestras pruebas no nos están ayudando. Al menos, no todo lo que deberían.

Lo que queremos comprobar en la prueba es, realmente, si hay una nueva tarea añadida. Una manera de probarlo es usar el método `pending_browns`. Uno podría pensar que no es una buena idea porque, si hay un error en `add_brown` y otro en `pending_browns` que se cancelen mutuamente, las pruebas pasarán igualmente. Eso es verdad, pero en la mayoría de los casos *no importa*, porque desde el punto de vista del usuario de la clase, ésta se comporta como debería. Cuando descubramos el fallo, lo podremos arreglar no sólo sin tener que cambiar las pruebas o el código que llama a `BrownList`, sino sin que haya habido ningún cambio en el comportamiento de `BrownList` desde el punto de vista de los usuarios.

```
class TestBrownList < Test::Unit::TestCase
  def setup
    # Recreamos la base de datos de pruebas para que esté vacía
  end

  def test_add_brown_simple
    db_handle = ...
    bl = BrownList.new(db_handle)
    bl.add_brown("Tarea de prueba")

    assert_equal 1, bl.pending_browns.length
  end
end
```

Mejor ejemplo de prueba del método `add_brown` de `BrownList`

Para terminar de ilustrar este consejo, imaginemos ahora que escribimos una interfaz web para nuestra aplicación de tareas pendientes. Si queremos comprobar que la interfaz web funciona correctamente, una (mala) idea que puede pasarnos por la cabeza es comparar el HTML de la página con el HTML que esperamos. Si comparamos el HTML completo (o una captura de pantalla), nuestras pruebas serán muy, muy frágiles. Por ejemplo, nuestras pruebas fallarán cuando hagamos cualquiera de estos cambios:

- Cambiar el id de algún elemento o el nombre de alguna clase CSS
- Cambiar un elemento de sitio o intercambiar la posición de dos opciones en un menú
- Añadir una nueva opción o información extra
- Corregir una falta de ortografía o redactar un texto de forma diferente

Si nuestras pruebas comparan la salida HTML exacta, implícitamente estamos definiendo nuestra aplicación no como una aplicación web con ciertas características, sino como una aplicación que genera ciertas cadenas de HTML. Ya que al usuario no le importa el HTML generado, sino que la aplicación funcione, podemos ver que este enfoque no es el más apropiado.

Una forma mucho mejor de probar una aplicación web es *buscar* las partes interesantes. Por ejemplo, comprobar que el título de la nueva tarea aparece en el contenido de la página justo después de crearla. O comprobar que ya no está ahí después de borrarla. O comprobar que, al renombrar una tarea, el título antiguo ya no aparece, pero sí el nuevo. Sin embargo, hacer esas comprobaciones directamente puede ser tedioso y puede añadir algo de fragilidad a nuestras pruebas, por lo que lo mejor es desacoplar los detalles del HTML generado de las comprobaciones que queremos hacer. Una de las técnicas para conseguir esto se conoce como *PageObjects*, pero explorar *PageObjects* va mucho más allá del objetivo de este artículo.

Como resumen de este consejo, podemos decir que las *pruebas no sólo deben fallar cuando hay algún problema, sino que también deben pasar mientras no haya ninguno*.

No ejecutarlas con frecuencia

Las pruebas no son un añadido al código, son parte integrante de éste. Asimismo, ejecutarlas es parte del ciclo normal de desarrollo. Si no las ejecutamos con frecuencia, no van a ser tan efectivas. Primero, porque cuando haya fallos, es probable que sea más de uno. En ese caso, será más difícil encontrar el origen de éstos. ¿Es un solo error el que provoca todos los fallos en las pruebas, uno por cada prueba? Segundo, porque si hemos hecho muchos cambios desde la última vez que ejecutamos las pruebas, tendremos más código que revisar en busca del problema.

Ejecutar las pruebas con frecuencia (idealmente, después de cada cambio que hacemos) hace que sea muy fácil encontrar la causa del error, porque lo único que puede haber sido la causa de los fallos son los cambios desde la última vez que las ejecutamos. Si ejecutamos las pruebas antes de mandar nuestros cambios al control de versiones, y vemos que una de las pruebas falla, será suficiente ejecutar `git diff` (o `svn diff` o similar) para ver qué cambios deben de haber producido el problema. Además, cuanto más alta sea la frecuencia con la que ejecutemos las pruebas, más seguros estaremos de que el código funciona correctamente. En la medida de lo posible, en el mundo de la programación es mejor evitar la fé: trabajaremos más tranquilos y con más confianza si podemos demostrar que el código funciona en los casos cubiertos por las pruebas.

El último punto importante de este consejo es tener una máquina «neutral» que ejecute las pruebas automáticas que tengamos, *cada vez que alguien manda un cambio al control de versiones*. Las ventajas son muchas:

- Incluso si alguien se olvida de ejecutar las pruebas antes de enviar los cambios, tenemos garantizado que las pruebas se ejecutarán.
- Si alguien se olvida de añadir un fichero al control de versiones, ese fichero no aparecerá en la máquina de integración continua, por lo que las pruebas fallarán y nos daremos cuenta del error.
- Los resultados de las pruebas en la máquina de integración continua son más fiables, porque tiene la misma configuración que las máquinas de producción. Por ejemplo, si un programador escribe una nueva prueba que depende de un nuevo módulo o de un cambio de configuración que sólo existe en la máquina de ese programador, la prueba pasará en su máquina, pero fallará en integración continua. Este fallo nos avisará del problema antes de que el proyecto pase a producción.
- Como tenemos los resultados de las pruebas para cada cambio que se haya hecho, y ejecutados en la misma máquina, podemos saber qué cambio exacto produjo el problema, lo que hace mucho más fácil arreglarlo.

Véase el artículo de Martin Fowler [fowlerci] sobre integración continua para más información.

No controlar el entorno

Otro problema bastante común es escribir pruebas sin controlar el entorno en el que se ejecutan. En parte esta (mala) costumbre viene de la creencia de que las pruebas tienen que adaptarse a diferentes circunstancias y ser robustas como los programas que escribimos. Esto es un malentendido.

Volvamos al ejemplo anterior de la aplicación de tareas pendientes. Cuando escribimos las pruebas, los pasos *no* fueron:

1. Obtener el número de tareas actuales, llamarlo n
2. Añadir una tarea
3. Comprobar que el número de tareas actuales es $n + 1$

Los pasos fueron:

1. Dejar la base de datos en un estado conocido (en este caso, vacía)
2. Añadir una tarea
3. Comprobar que el número de tareas es *exactamente* 1

Esta diferencia es fundamental. Uno podría pensar que la primera prueba es mejor porque «funciona en más casos». Sin embargo, esto es un error por las siguientes razones:

- Escribir código robusto requiere mucho más esfuerzo mental, especialmente a medida que crecen las posibilidades. Como no necesitamos esa robustez, mejor dejarla de lado.
- Las pruebas serán menos flexibles, porque no podremos probar qué ocurre en casos específicos (p.ej. cuando hay exactamente 20 tareas, cuando hay más de 100 tareas, etc.).
- Si no controlamos y rehacemos el entorno de ejecución de las pruebas, unas pruebas dependerán, potencialmente, de otras. Lo que significa que el comportamiento de unas pruebas puede cambiar el resultado de otras. En el caso ideal, que es el caso común, las pruebas se pueden ejecutar una a una independientemente y tienen exactamente el mismo resultado.
- No siempre darán el mismo resultado, incluso cuando las ejecutemos por sí solas. Por ejemplo, digamos que hay un fallo en `add_brown` que sólo aparece cuando hay más de 20 tareas. En ese caso, si nunca borramos la base de datos, nuestras pruebas fallarán... cuando las hayamos ejecutado suficientes veces. Y si las dejamos así, y hay otro fallo que sólo aparece cuando no haya ninguna tarea, las pruebas nunca nos avisarán del segundo fallo.

Si queremos probar ciertos casos de datos iniciales, es más claro y más fiable probar esos casos expresamente y por separado. Tendremos la ventaja de que estará claro al leer las pruebas qué casos cubrimos, y ejecutar las pruebas *una sola vez* nos hará estar seguros de que *todos* los casos que nos interesan funcionan perfectamente. Como regla general, cualquier incertidumbre o indeterminismo sobre la ejecución o resultados de las pruebas que podamos eliminar, debe ser eliminado.

Podemos terminar este consejo con una reflexión: *las pruebas no son mejores porque pasen con más frecuencia, sino porque demuestren que un mayor número de casos interesantes funcionan exactamente como queremos*.

Reusar datos de prueba

Cuando empezamos a escribir pruebas, algo que necesitamos con frecuencia son datos iniciales o de prueba (en inglés, *fixtures*). Si no tenemos una forma fácil de crear esos bancos de datos para cada prueba, tendremos la tentación de tener un solo conjunto de datos iniciales que usaremos en *todas* las pruebas de nuestro proyecto. Aunque en algunos casos pueda resultar práctico compartir datos de prueba entre *algunas* pruebas, esta costumbre puede traer un problema añadido.

A medida que escribimos nuevas pruebas, éstas necesitarán más datos de contraste. Si añadimos estos datos a nuestro único conjunto de datos iniciales, cabe la posibilidad de que algunas de las pruebas antiguas empiece a fallar (p.ej. una prueba que cuente el número de tareas en el sistema). Si ante este problema reescribimos la prueba antigua para que pase con el nuevo conjunto de datos, estaremos haciendo más complejas nuestras pruebas, y además corremos el riesgo de cometer un fallo al reescribir la prueba antigua. Por no mencionar que si seguimos por este camino, puede que en la siguiente ocasión tengamos que reescribir dos pruebas. O cinco. O veinte.

Todo esto está relacionado, en cierta manera, con el problema descrito en el anterior apartado: pensar en las pruebas como pensamos en el resto del código. En este caso, pensar que tener más datos de prueba es mejor, porque se parecerá más al caso real en el que se ejecutará el programa. Sin embargo, en la mayoría de los casos esto no representa ninguna ventaja, pero sí que tiene al menos una desventaja: cuando alguna prueba falle y tengamos que investigar por qué, será más difícil encontrar el problema real cuantos más datos haya. Si podemos escribir nuestras pruebas teniendo un solo objeto de prueba, o incluso ninguno, mejor que mejor.

No facilitar el proceso de pruebas

El apartado sobre ejecutar las pruebas con frecuencia ya mencionaba que las pruebas son parte integrante del código. Aunque no

funciona exactamente de la misma manera ni tienen las mismas propiedades, sí que se tienen que mantener con el mismo cuidado y esfuerzo con el que mantenemos el resto del código.

Este apartado hace hincapié en que tenemos que hacer lo posible para facilitar la escritura de pruebas. Éstas no son una necesidad molesta a la que tenemos que dedicar el menor tiempo posible: como parte integrante de nuestro código se merece la misma dedicación que el resto. Así, nuestro código de prueba debe ser legible, conciso y fácil de escribir. Si cuesta escribir pruebas, ya sea en tiempo, esfuerzo mental o líneas de código, tenemos un problema que debemos resolver, ya sea reorganizando código, escribiendo métodos de conveniencia o usando cualquier otra técnica que nos ayude. Desgraciadamente, muchos desarrolladores piensan que es normal que sea costoso escribir pruebas y no hacen nada por mejorar la situación. En última instancia, esto hace que el equipo escriba menos pruebas, y de peor calidad.

Veamos un caso concreto. Digamos que queremos probar la interfaz web de nuestra aplicación de tareas pendientes. Una de las primeras pruebas que escribiríamos aseguraría que crear una tarea simple funciona. Una primera implementación podría quedar así:

```
class TestBrownListDashboard_____MAL(unittest.TestCase):

    def setUp(self):
        # Rehacemos la base de datos y creamos el navegador en self.driver

    def testAddBrownSimple_____MAL(self):
        self.driver.get("/")
        self.driver.findElementById("username").send_keys("usuario")
        self.driver.findElementById("password").send_keys("contraseña")
        self.driver.findElementById("login").click()

        new_brown_title = "My title"
        self.driver.findElementById("new_brown").send_keys(new_brown_title)
        self.driver.findElementById("create_brown").click()
        title_tag = self.driver.findElementByTagName("task-1-title")
        self.assertEqual(title_tag.text, new_brown_title)
```

Ejemplo de prueba funcional difícil de escribir

Aunque aisladamente, este código es relativamente fácil de leer y entender, tiene varios problemas:

- No es todo lo compacto que podría ser
- Contiene código que sabemos que se duplicará en otras pruebas
- No contiene abstracciones, por lo que cuando haya cambios en la aplicación (digamos, el id de "username" o "password" cambia), tendremos que buscar dónde nos referimos a éstos para actualizar el código
- No está escrito usando el lenguaje del dominio de la aplicación, sino usando el lenguaje de automatización de un navegador, por lo que es más difícil de leer y mantener

Una alternativa mucho mejor sería la siguiente:

```
class TestBrownListDashboard(BrownFunctionalTestCase):

    def testAddBrownSimple(self):
        self.assertLogin("usuario", "contraseña")

        new_brown_title = "My title"
        self.createBrown(new_brown_title)
        self.assertBrownExists(new_brown_title)
```

Ejemplo de prueba funcional más fácil de escribir

Las mejoras de la segunda versión son las siguientes:

- `TestBrownListDashboard` ahora hereda de una nueva clase, `BrownFunctionalTestCase`, que será una clase base para todas nuestras clases de prueba. Aquí añadiremos todo el código común a diferentes pruebas de nuestra aplicación.
- Como tenemos una clase base, ya no necesitamos escribir el método `setUp` porque ésta ya crea la base de datos e inicializa el navegador de prueba por nosotros.
- Para abrir sesión, simplemente llamamos a un nuevo método `assertLogin`. No sólo es mucho más compacto y legible, sino que si alguna vez cambian los detalles de cómo iniciamos sesión, podemos simplemente cambiar la implementación de este método.
- Crear una nueva tarea es tan fácil como llamar a un nuevo método `createBrown`, y comprobar que se ha creado correctamente se lleva a cabo llamando al método `assertBrownExists`. Dependiendo del caso, podríamos incluso haber creado un método `assertCreateBrown`, pero por ahora parece mejor dejar ambas operaciones separadas.

Como se puede ver, una simple reorganización del código (del mismo tipo que haríamos con el código principal del programa) puede tener un impacto muy grande en la facilidad de mantenimiento de nuestras pruebas.

La necesidad de facilitar la escritura de pruebas se extiende a todas las tareas relacionadas con probar nuestro código, no sólo mantener el código de pruebas automáticas. Digamos que escribimos un programa cliente-servidor. Si cada vez que encontramos un problema no somos capaces de depurarlo, o de asegurar si está de verdad arreglado o no porque no tenemos una forma fácil de probar el cliente o el servidor por separado, tenemos un problema. Una de las varias soluciones posibles es tener un cliente de prueba con el que podamos enviar al servidor cualquier petición que se nos ocurra, y un servidor de prueba con el que podamos enviar al cliente cualquier respuesta que se nos ocurra. Herramientas para capturar *fácilmente* el tráfico entre cliente y servidor también pueden ahorrarnos mucho tiempo a la larga.

Al fin y al cabo, estamos hablando de la calidad de nuestro trabajo, no en un sentido abstracto o teórico, sino en el sentido más pragmático desde el punto de vista del usuario. Si no podemos comprobar que nuestro programa se comporta debidamente, nos quedarán muchos fallos por descubrir, y por tanto que arreglar, que llegarán a los usuarios finales.

Depender de muchos servicios externos

El último consejo es el más avanzado, y es el consejo con el que hay que tener más cuidado al aplicarlo. La tendencia natural al crear entornos de prueba es replicar algo lo más parecido posible al entorno real, usando las mismas bases de datos, los mismos servidores y la misma configuración. Aunque esto tiene sentido y es *necesario* en pruebas de aceptación y pruebas de integración [1], puede ser bastante contraproducente en pruebas unitarias y similares, en las que sólo queremos probar componentes relativamente pequeños.

Depender de servicios externos como una base de datos, un servidor web, una cola de tareas, etc. hace que las pruebas sean más frágiles, porque aumentan las posibilidades de que fallen por una mala configuración, en vez de porque hemos encontrado un problema en el código. Como en la mayoría de los tipos de pruebas, como las unitarias, sólo queremos probar que cierto componente concreto funciona correctamente, no hace falta que lo integremos con el resto de los componentes. En muchos casos, podemos reemplazar esos componentes con versiones «de pega» que se comporten como nos haga falta para cada prueba.

Un ejemplo claro de las ventajas de usar componentes «de pega» es el desarrollo y prueba de una aplicación que usa una API, tanto en el caso de que escribimos sólo el cliente como en el caso de que escribimos tanto el cliente como el servidor. Aunque para hacer pruebas de integración deberíamos usar el servidor real, la mayoría de las pruebas tendrán un ámbito más limitado (bien sean pruebas unitarias, o pruebas funcionales que sólo cubran el comportamiento del cliente). Para éstas sabemos, al tener la documentación de dicha API:

1. Qué llamadas debe generar nuestra aplicación en ciertas situaciones.
2. Cómo tiene que reaccionar nuestra aplicación ante ciertas respuestas del servidor.

Armados con este conocimiento, podemos diseñar pruebas que no dependen del servidor. No depender del servidor tiene varias ventajas, entre otras:

- Las pruebas se ejecutan más rápido.
- Las pruebas se pueden ejecutar sin tener conexión a internet o acceso al servidor (incluyendo que el servidor esté funcionando correctamente).
- Cuando hay fallos, éstos son más fáciles de depurar y corregir.
- Podemos probar muchas situaciones que no podemos reproducir fácilmente en un servidor real, como no poder conectar al servidor, que el servidor devuelva cualquier tipo de error, que devuelva ciertas combinaciones de datos que son difíciles de obtener, condiciones de carrera, etc.

No todo son ventajas, claro. Si el servidor introduce cambios que rompen la compatibilidad con la documentación que tenemos, esos fallos no se descubrirán hasta las pruebas de integración. De manera similar, si nuestras pruebas dependen de un comportamiento concreto, documentado o no, y este comportamiento cambia, de nuevo no detectaremos estos fallos hasta las pruebas de integración.

Veamos un ejemplo más concreto. Digamos que escribimos un programa que utiliza la API pública de Kiva, una organización que permite, mediante microcréditos, prestar dinero a personas que lo necesitan. Nuestra aplicación mostrará los últimos préstamos listados en la web de Kiva (digamos, 50), información que obtenemos usando la llamada `/loans/newest`. Sin embargo, hay varios casos que son muy difíciles de probar con un servidor real:

- Esa funcionalidad de la API sólo devuelve 20 elementos por llamada, por lo que tenemos que hacer varias llamadas para obtener todos los préstamos que queremos.
- Si se añaden nuevos préstamos mientras estamos haciendo las llamadas para obtener 50 préstamos, tendremos préstamos repetidos, lo que queremos evitar.
- Puede ocurrir que no haya 50 préstamos en un momento dado, así que tendremos que conformarnos con los datos que haya (en vez de, por ejemplo, entrar en un bucle infinito).
- Puede que el servidor tenga algún problema y no devuelva una respuesta correcta, o que simplemente haya problemas de conectividad entre el cliente y el servidor. Como mínimo queremos asegurarnos, como en el caso anterior, de que el cliente no se queda en un bucle infinito, haciendo peticiones ciegamente hasta que hayamos recibido 50 préstamos.

Probar todos esos casos a mano con el servidor real de Kiva es prácticamente imposible, principalmente porque no podemos hacer que el servidor devuelva las respuestas necesarias para reproducir cada caso. Si todas nuestras pruebas dependen del servidor no podremos estar seguros de si el código funciona bien. Sin embargo, todos estos casos son muy fáciles de probar si evitamos conectarnos al servidor real. Los casos mencionados arriba se podrían escribir de la siguiente manera en Javascript, usando Jasmine:

```
it("should correctly get items from several pages", function() {
  var fakeServer = new FakeServer(fixtureWith100Loans);
  var kivaLoanLoader = new KivaLoanLoader(fakeServer);
  kivaLoanLoader.fetchLoans(50);
  expect(kivaLoanLoader.loans.length).toEqual(50);
  expect(kivaLoanLoader.loans[0].id).toEqual("loan1");
  expect(kivaLoanLoader.loans[50].id).toEqual("loan50");
});

it("should correctly skip items duplicated in different pages", function() {
  var fakeServer = new FakeServer(fixtureWith100LoansSomeRepeated);
  var kivaLoanLoader = new KivaLoanLoader(fakeServer);
  kivaLoanLoader.fetchLoans(25);
  expect(kivaLoanLoader.loans.length).toEqual(25);
  expect(kivaLoanLoader.loans[19].id).toEqual("loan20");
  // El siguiente caso será loan20 repetido, si el código no funciona bien
  expect(kivaLoanLoader.loans[20].id).toEqual("loan21");
  expect(kivaLoanLoader.loans[24].id).toEqual("loan25");
});

it("should stop when there's no more data", function() {
  var fakeServer = new FakeServer(fixtureWith30Loans);
```

```
  var kivaLoanLoader = new KivaLoanLoader(fakeServer);
  // La línea siguiente será un bucle infinito si el código no es correcto
  kivaLoanLoader.fetchLoans(40);
  expect(kivaLoanLoader.loans.length).toEqual(30);
  expect(kivaLoanLoader.loans[0].id).toEqual("loan1");
  expect(kivaLoanLoader.loans[29].id).toEqual("loan30");
});

it("should stop on server errors", function() {
  var fakeServer = new FakeServer(fixtureWithOnlyServerError);
  var kivaLoanLoader = new KivaLoanLoader(fakeServer);
  // La línea siguiente será un bucle infinito si el código no es correcto
  kivaLoanLoader.fetchLoans(20);
  expect(kivaLoanLoader.loans.length).toEqual(0);
});
```

Ejemplo de cómo probar el cliente ficticio de la API de Kiva

Conclusión

Probar programas es una tarea importante pero bastante difícil de hacer correctamente. Sin embargo, si empezamos a hacer pruebas desde el comienzo del proyecto, las ejecutamos con frecuencia y nos preocupamos de que sean fáciles de mantener, nuestras probabilidades de producir programas robustos serán mucho más altas.

Calidad en software

Esteban Manchado Velázquez

«Calidad» es una palabra muy usada pero un concepto bastante escurridizo, al menos a la hora de encontrar maneras fiables de mejorarla o simplemente mantenerla una vez hemos llegado a un nivel aceptable. Este artículo explorará qué es la calidad y cómo mejorarla en nuestros proyectos.

Significado

El primer problema de la calidad es definirla. Hay muchas definiciones, pero mi preferida en el contexto de la ingeniería de software es «adecuación al uso». Uno de los problemas de la definición es que es muy vaga pero, paradójicamente, el propio hecho de ser tan vaga es lo que la hace útil. La calidad es un asunto muy complejo, por lo que simplificarlo, lejos de ayudarnos a entender, sólo nos dará la ilusión de que lo entendemos. Y la ilusión de entendimiento es muy peligrosa porque hace que nos resistamos al aprendizaje real.

El segundo problema de la calidad es hacer que todos los interesados compartan lo que entienden por ésta. Este entendimiento compartido nos ayudará a centrarnos en los objetivos importantes, que son las necesidades del cliente del proyecto. Esto no significa que sólo sean importantes los aspectos que el cliente *menciona*: con frecuencia, los clientes dan por sentadas una serie de propiedades (eficiencia, fiabilidad, etc), y nuestro trabajo es asegurarnos de que éstas se cumplen tanto como los requisitos explícitos. Esto es, la calidad necesita de aspectos técnicos como código rápido o diseño simple, pero éstos deben estar supeditados a las necesidades y los deseos del cliente, implícitos y explícitos.

Cómo mejorar la calidad

Como ya expone el apartado anterior, es imposible dar una «receta mágica» para mejorar la calidad. De hecho, intentar mejorar la calidad simplemente siguiendo una lista de pasos es un camino casi seguro hacia el fracaso. No importa qué pasos sigamos, qué hayamos oído de ellos o quién los haya recomendado: nuestra única vía para alcanzar un buen nivel de calidad es usar nuestra experiencia y conocimiento del contexto para decidir qué es lo que nos ayudará en cada momento. Tenemos que ser conscientes de lo que hacemos y tomar el control de nuestras decisiones.

Sin embargo, sí se pueden dar *guías* para mejorar la calidad. Por ejemplo: mantener siempre el qué, no el cómo, como la guía de todo lo que hacemos; mantener el escepticismo y cuestionar cómo hacemos las cosas y por qué; estar preparado para cambiar cómo trabajamos y luchar contra la «programación de culto al cargo» [2]; no creer en la tecnología como el centro de lo que hacemos; darse cuenta de que lo principal no es hacer código bonito o fácil de entender, sino resolver problemas [3]; no creer que los problemas tengan una solución única o mejor.

Esto no quiere decir que las herramientas, técnicas y costumbres no sean útiles. En absoluto. Lo que sí significa es que éstas nos ayudarán según qué contextos, según qué proyectos, según qué equipos y según qué clientes, pero nunca en todos los casos. Los otros artículos de este libro describen algunas de estas técnicas y herramientas, que son muy útiles y todo buen profesional debe conocer y dominar, pero uno de los mensajes principales de este artículo es que conocer y adaptarse al contexto es importante, y que aplicar ciegamente cualquiera de estas técnicas o herramientas es un error.

Para ilustrar las ideas de este artículo, y como inspiración para ayudar a pensar fuera de cánones más académicos, los siguientes apartados muestran una serie de ejemplos de situaciones junto con sugerencias de posibles soluciones. Obviamente no puede haber solución única o «correcta» en estos ejemplos, entre otras razones porque ninguna descripción literaria puede dar toda la información necesaria para tomar una buena decisión.

Ejemplo 1: procesos

Como regla general, tener procesos y reglas ayuda a los equipos de desarrollo a trabajar más eficientemente. Por una parte, facilita que nos concentremos en lo que estamos haciendo y no en el cómo [4]. Por otra, facilita la automatización.

Sin embargo, los procesos y las reglas pueden quedarse obsoletos. Por ejemplo, digamos que uno o dos miembros del equipo causan problemas con la integración continua: con frecuencia hacen cambios que hacen fallar a la batería de pruebas y no tienen disciplina para escribir pruebas. Ante esta situación, decidimos que cada cambio enviado al repositorio debe contener alguna modificación a

algún fichero de pruebas. La medida funciona más o menos bien, esos miembros del equipo empiezan a tomarse las pruebas más en serio, y así aumentamos la productividad del equipo con ella.

Ahora bien, todas las reglas tienen partes buenas y partes malas, como todo. Esta medida concreta puede ser molesta cuando simplemente queramos arreglar una falta de ortografía en un comentario, actualizar la documentación, o reformatear el código (ya que nuestro cambio no modificaría ningún fichero de pruebas). Por tanto, si en algún momento esos miembros del equipo se marchan o llegan al punto de escribir buenas pruebas y no necesitar la anterior medida, puede que llegue el momento de eliminar esta regla. Mantener y seguir reglas simplemente porque «siempre han funcionado» es caer en la tentación de seguir el «culto al cargo».

Ejemplo 2: automatización de pruebas

Una de las constantes en los artículos de este libro es el uso de pruebas automáticas. Aunque es desde luego una de las prácticas más recomendadas y útiles, también es verdad que con frecuencia tenemos que trabajar con código legado (es decir, sin pruebas). En esos casos, ¿qué hacemos?

Por ejemplo, digamos que nos unimos a un equipo que no tiene cultura de pruebas automáticas, y vamos retrasados con la siguiente entrega. Una parte concreta del código está dando bastantes problemas, y las pruebas manuales (la manera en la que el equipo prueba el código) no están encontrando suficientes fallos, o no lo suficientemente pronto. Aunque automatizar las pruebas es probablemente la mejor idea a largo plazo, no hay ninguna regla exenta de excepciones [5]. Puede que el equipo no tenga suficiente experiencia en pruebas automáticas como para que introducirlas mejore la situación *antes de la entrega*. Puede que las partes críticas del proyecto en el que trabajamos no sean fáciles de probar de manera fiable con pruebas automáticas, y empeñarnos en poner énfasis en éstas a toda costa no ayude al equipo en el contexto de esta entrega. Puede que la tensión de ir retrasados haga al equipo ser «optimistas» al escribir pruebas, y las pruebas den un falso sentido de seguridad. Puede que los que toman las decisiones no estén convencidos, y que a corto plazo no valga la pena gastar tiempo y energía en convencerlos. Y puede que hayamos convencido a los que toman las decisiones, pero *el equipo* no esté convencido, e intentar forzarlos a escribir pruebas sólo vaya a provocar un bajón de moral y un conjunto de pruebas automáticas de muy mala calidad.

Y si por la razón que sea llegamos a la conclusión de que intentar implantar pruebas automáticas para la próxima entrega no es una buena idea, ¿qué podemos hacer? Una de las posibilidades es empezar haciendo más efectivo al equipo sin cambiar radicalmente su filosofía de trabajo. Por ejemplo, puede que tras analizar la situación descubramos que esa parte del programa es más difícil de probar de manera manual porque no da suficiente información. Quizás añadir una opción «escondida» que haga esa parte menos opaca puede ser suficiente hasta la fecha clave. O simplemente mejorar la comunicación entre los miembros del equipo. Porque, entre otras cosas, siempre es positivo respetar la manera de trabajar de un equipo («que siempre ha funcionado»): no sólo mejora las relaciones entre sus miembros, sino que ayuda a ganarse su respeto y atención, que serán necesarios más tarde para poder implementar cambios grandes como desarrollo dirigido por pruebas o simplemente escribir pruebas automáticas. Y mientras tanto, podemos ir enseñando poco a poco al equipo a automatizar las pruebas para ir cambiando a un modelo (posiblemente) más escalable.

Ejemplo 3: especificaciones

Cuando se desarrolla software normalmente se tienen especificaciones. Pueden ser formales (un documento) o informales (el conocimiento compartido del equipo). El objetivo de las especificaciones es poder concentrarse en la solución técnica sin tener que estar preguntando continuamente a los clientes o al resto del equipo sobre el enfoque a la hora de resolver el problema. Pero, de nuevo, las especificaciones son sólo una herramienta, y cumplir más a rajatabla con las especificaciones no tiene por qué garantizar una mayor calidad del proyecto una vez completado.

Digamos que estamos desarrollando el software que conduce un coche automático. Uno de los requisitos del coche es que pare en los pasos de peatones cuando detecte que una persona está cruzando. Sin embargo, mucha gente no cruza por los pasos de peatones, así que el coche sería mucho más seguro si no dependiera de éstos, sino que pudiera detectar por sí mismo si tiene algo delante que podría llegar a atropellar. Es decir, las especificaciones son una herramienta muy útil, pero nunca son el objetivo final del desarrollo de software. Las personas que escriben las especificaciones cometen errores, las condiciones del proyecto cambian, etc. Mantener siempre el escepticismo y una visión más completa de nuestros objetivos, y no dejarnos llevar por el «no es mi trabajo», el «no me pagan por esto» o el «yo sólo hago lo que me dicen», es mucho más importante que cumplir la especificación diligentemente. Dicho de otra manera, nuestro trabajo no es escribir software que cumple a rajatabla una descripción textual de un problema. Es hacer software útil y resolver problemas.

Ejemplo 4: diseño simple

Una parte importante de la calidad es, por supuesto, tener código mantenible. El código mantenible normalmente se consigue con código legible y un diseño simple. Sin embargo, y como muchas otras cosas, estos dos aspectos son sólo una herramienta para conseguir calidad: un código legible y un diseño simple hacen que el código contenga, de media, menos errores, y éstos serán más fáciles de detectar y corregir.

Ahora, ¿qué pasa si en algún momento las necesidades del proyecto chocan con nuestro (por ahora) diseño simple? La respuesta es obvia: las necesidades del cliente son el objetivo número uno, y lo demás se tiene que adaptar a éstas. Intentar adaptar las necesidades del cliente al diseño de la aplicación es, en la mayoría de los casos, un error. Si para resolver el nuevo problema hacemos el diseño menos lineal o más complejo, no estamos «haciendo una chapuza porque el cliente no tiene las ideas claras» o porque «no sabe cómo funciona la aplicación»: estamos ayudando a resolver un problema real. Si eso implica hacer una «chapuza» en el código, eso probablemente significa que tenemos que revisar el diseño de nuestra aplicación. No porque lo hayamos hecho mal desde el principio, sino porque hemos descubierto nuevos requisitos, o refinado los que teníamos.

Conclusiones

Una conclusión a la que podemos llegar es que la calidad es difícil de conseguir y de medir, y se necesita experiencia y mucho trabajo para obtenerla. Pero la conclusión más importante es que *es imposible mejorar la calidad de un proyecto informático aplicando reglas o metodologías*. Da igual cuánta experiencia o cuánto conocimiento tenga la persona que las haya formulado, ningún conjunto de reglas o metodologías puede resolver nuestros problemas si las aplicamos sin entender lo que hacemos y en qué contexto son útiles.

Integración continua

Yeray Darias Camacho

Independientemente de si el equipo de desarrollo sigue una metodología clásica en cascada o algún tipo de metodología ágil hay un momento decisivo que determina el éxito del proyecto. Este momento es el despliegue de la aplicación en los sistemas del cliente, lo que conocemos como sistema de producción.

Generalmente este suele ser un momento muy tenso porque es muy raro que todo funcione a la primera. Si se sigue una metodología en la que predomine el desarrollo incremental, donde las funcionalidades se van entregando poco a poco, se minimiza ligeramente el impacto, siempre y cuando al final de cada iteración se haya desplegado en el sistema de producción real. Pero generalmente sigue siendo un momento incómodo, se suelen producir errores porque las máquinas de desarrollo tienen configuraciones diferentes a las máquina de producción, el rendimiento no es tan bueno porque la base de datos de producción tiene una cantidad mucho mayor de información, o cualquier otro detalle que no se tuvo en cuenta durante el desarrollo.

Para resolver este problema aparece una nueva «filosofía» o práctica denominada integración continua. Es un modo de desarrollar un poco diferente al habitual, y que requiere de una serie de buenas prácticas y la aceptación de las mismas por el equipo de desarrollo. Se ha de convertir en un hábito que se realice de forma automática, casi sin darnos cuenta.

La integración continua desde el punto de vista del desarrollador

La siguiente descripción de un día de trabajo, en un equipo que realiza integración continua, ayudará a ilustrar el proceso y a comprender los elementos necesarios para llevarla a cabo.

Al principio del día lo normal es seleccionar la siguiente tarea más importante a realizar. En base a la reunión de planificación [6] y a la reunión diaria [7] siempre existe una lista de tareas priorizadas a disposición del equipo de desarrollo, por lo que es muy sencillo saber qué es lo siguiente que debemos hacer. Seleccionamos la tarea en la que debemos trabajar y volvemos a nuestra mesa de trabajo.

El primer paso será actualizar el código fuente del que disponemos con la versión más nueva que exista en el repositorio central. De igual manera, si fuese la primera vez que vamos a trabajar en un proyecto, no tenemos más que descargar una copia limpia del repositorio de código y empezar a hacer nuestro trabajo. A lo largo del día implementaremos la nueva funcionalidad, que debería ser bastante pequeña como para terminarla en una jornada de trabajo y debería incluir una serie de tests que verifiquen que posee el comportamiento deseado. Se puede leer más sobre los tests en otros capítulos del libro, ahora mismo no ahondaremos en el tema porque existen libros enteros sobre TDD y tests unitarios.

Cuando la funcionalidad esté terminada, antes de subir ningún cambio al repositorio, actualizaremos el código fuente con los cambios de nuestros compañeros y nos aseguraremos de que la aplicación sigue construyéndose correctamente y que los tests del proyecto están en verde, es decir que pasan todos sin ningún problema. Si por el contrario aparece algún error lo arreglaremos inmediatamente. Nunca, bajo ningún concepto, se debe subir código al repositorio sin revisar que los tests pasan correctamente y la aplicación se puede construir sin incidencias. Además, es recomendable acceder a la aplicación y revisar rápidamente que todo sigue funcionando adecuadamente, ya que por lo general los tests unitarios no tienen una cobertura del 100%, puesto que ciertos detalles de infraestructura son más sencillos de probar a mano.

Una vez hemos realizado todos los pasos anteriores podemos guardar nuestros cambios en el repositorio de código central, lo que permite que el resto del equipo se actualice y los tengan disponibles en cuestión de segundos.

Aunque el proceso de integración continua comenzó desde el momento en el que empezamos a trabajar en la nueva funcionalidad, el servidor de integración continua comienza a trabajar cuando subimos nuestros cambios al repositorio de código. Dicho sistema se descargará una versión nueva del código fuente con todos los cambios realizados (los nuestros y los de nuestros compañeros), ejecutará los tests y tras hacer la construcción del proyecto lo desplegará en una «réplica» de la máquina de producción. Todo de forma totalmente automatizada. El propio servidor de integración continua podría pasar algunas pruebas extras, como por ejemplo análisis estático de código, análisis de cobertura, o cualquier otro detalle que sería muy tedioso pasar en el proceso de desarrollo porque requiere demasiado tiempo.

También podría haber ocurrido un error. En ese caso el servidor de integración continua nos avisaría con algún mensaje en la consola del mismo o simplemente con el envío de un correo electrónico, que es la opción más común. En ese caso tendremos que inspeccionar cuál ha sido el error y resolverlo para subir una nueva versión corregida. Como el error se genera a los pocos minutos, y no han pasado días ni meses, es muy fácil encontrar dónde está el problema.

Ventajas de la integración continua

La experiencia demuestra que reduce de manera increíble el número de errores en el producto final. Esta es probablemente la mayor ventaja que aporta, porque un producto final con pocos o incluso ningún error es a fin de cuentas el objetivo que todos deseamos como desarrolladores.

Pero no hay que olvidarse de la otra gran ventaja que aporta esta práctica, la transparencia del proceso. Como todos trabajamos utilizando el mismo repositorio de código y la información del servidor de integración es pública, para todo el equipo e incluso a veces para el cliente, se conoce con precisión el estado real del proyecto. No hay que esperar durante meses para saber cómo van las cosas y las reuniones de seguimiento pueden ser cortas y precisas.

Como ventaja añadida cabe destacar la posibilidad de disponer de un servidor de demostración en el que siempre se posee la última versión de la aplicación. De esta manera los clientes pueden probar los últimos cambios día a día y ofrecer información al equipo de desarrollo sin tener que esperar meses.

Requisitos de la integración continua

Ahora ya disponemos de una visión más concreta de lo que es la integración continua, pero nos vendrá bien repasar cuáles son los elementos indispensables para hacerlo correctamente.

Repositorio de código

El repositorio de código es fundamental la herramienta que permite que el equipo trabaje de forma totalmente sincronizada, pero a

la vez sencilla. Cada uno trabaja en su parte y puede actualizar los cambios sin necesidad de que otros desarrolladores tengan que esperar por estos y viceversa.

Desgraciadamente todavía hay muchas empresas que aún no poseen una herramienta de este tipo. Para aquellos que aún no utilicen un repositorio de código cabe destacar que actualmente existen una gran cantidad de soluciones gratuitas de calidad, como pueden ser Subversion, Git o Mercurial por mencionar algunas. Esta es una pieza fundamental y será prácticamente imposible realizar integración continua si no se dispone de ella.

Este tipo de herramientas marcan un antes y un después en el trabajo cotidiano de un equipo de desarrollo de software. Pero además es el pivote fundamental de la integración continua, que no es posible sin un repositorio de código.

Sistema de construcción automatizado

Muchos equipos utilizan complejos entornos de desarrollo para implementar y compilar los proyectos. Esto en sí mismo no es malo, pero debemos poder construir el proyecto en cualquier máquina sin necesidad de dichos entornos. No todas las máquinas en las que vayamos a instalar nuestro proyecto tendrán el entorno de desarrollo con el que trabajamos, incluso en algunos casos estos pueden tener licencias relativamente caras, por lo que no sería una opción viable.

Hay muchas soluciones de construcción diferentes en función de los lenguajes de programación, en Java están Ant o Maven, en Ruby está Rake, solo por mencionar algunas. La condición es que debe ser una herramienta muy sencilla que se pueda ejecutar en cualquier máquina y que consiga automatizar no solo el proceso de compilación, sino la ejecución de los tests de nuestra aplicación e incluso el despliegue en el servidor de aplicaciones llegado el caso.

Commits diarios

No todos los requisitos se basan en tener un determinado tipo de herramienta, algunos se basan en saber cómo utilizar una determinada herramienta. Una vez que tenemos un control de versiones hay que recordar que el almacenamiento de los cambios debe ser diario, idealmente un desarrollador debe hacer incluso varias subidas al repositorio de código al día.

Si tenemos un repositorio de código, pero cada desarrollador sube sus cambios cada dos semanas seguimos en el mismo problema de antes, la integración del proyecto se retrasa durante todo ese tiempo, perdiendo toda opción de obtener información inmediatamente. Además, retrasar la integración tanto tiempo genera más conflictos entre los ficheros editados por los desarrolladores. Debido a que con cada actualización del repositorio se hace una integración del proyecto completo, cada mínimo cambio que no desestabilice el proyecto debe ser subido para que se compruebe que todo sigue funcionando correctamente.

Pruebas unitarias

Hemos hablado todo el tiempo de comprobar que el proyecto funciona correctamente y al mismo tiempo de automatizar todo el proceso lo mejor posible. Para poder comprobar que el proyecto funciona de forma automática necesitamos pruebas unitarias (muchas veces incluso pruebas de integración).

Existen muchas maneras diferentes de incluir tests en el proyecto, personalmente creo que TDD es la manera más adecuada, pero si el equipo no tiene experiencia y crea las pruebas después de haber realizado la implementación tampoco es un método inválido. La premisa fundamental es que toda nueva funcionalidad debe tener una batería de pruebas que verifique que su comportamiento es correcto.

Servidor de integración

Esta es la pieza más polémica, mucha gente cree que no es absolutamente necesaria para hacer integración continua correctamente, por ejemplo, algunos equipos hacen la integración de forma manual con cada subida al repositorio de código. En mi opinión es un paso tan sencillo y barato de automatizar que no merece la pena ahorrárselo. Montar algún servidor de integración, como por ejemplo Jenkins o Cruise Control, es gratuito y tan sencillo como desplegar un fichero en un servidor. Por contra las ventajas son grandísimas. Para empezar el proceso está totalmente automatizado, lo que evita el error humano. Y por otro lado reduce la cantidad de trabajo, ya que tenemos una solución prefabricada sin necesidad de tener que crear nosotros mismos complejas soluciones caseras.

Un paso más allá

Con los pasos descritos hasta el momento ya tendríamos un proceso bastante completo y que a buen seguro mejorará enormemente la calidad de nuestro producto. Pero podemos ir un poco más allá y utilizar el servidor de integración para que haga ciertas tareas relativamente complejas por nosotros.

Por ejemplo, podríamos configurar el servidor para que haga análisis estático de código, de forma que pueda buscar en todo el código bloques sospechosos, bloques duplicados o referencias que no se utilizan, entre otras cosas. Existen gran cantidad de opciones como pueden ser FindBugs o PDM. A simple vista puede parecer algo irrelevante, pero hay que recordar que la complejidad es lo que más ralentiza el proceso de desarrollo, por lo que un código con menor número de líneas y referencias inútiles será más sencillo de leer y entender.

También podríamos incluir tareas que permitan desplegar la aplicación en un servidor, de forma que tendríamos siempre un servidor de demostración con la última versión de nuestro proyecto. Esto es realmente útil cuando tenemos un cliente comprometido, dispuesto a probar todos los nuevos cambios y a dar información al equipo.

Otra operación que podemos automatizar, y que el servidor de integración podría hacer por nosotros, es la realización de pruebas de sistema sobre la aplicación. Imaginemos que estamos desarrollando una aplicación web, podríamos crear tests con alguna herramienta de grabación de la navegación, como por ejemplo Selenium, y lanzarlos con cada construcción que haga el servidor. Es un tipo de prueba que requiere mucho tiempo y no sería viable que se lancen con cada compilación del desarrollador, pero para el servidor de integración no habría ningún problema. Este es solo un ejemplo más de la cantidad de cosas que puede hacer un servidor de integración continua por nosotros, y que nos ayudará a mantener un producto estable y testeado de manera totalmente automática.

Para acabar me gustaría utilizar algunos comentarios escuchados por Martin Fowler cuando habla de integración continua con otros desarrolladores. La primera reacción suele ser algo como «eso no puede funcionar (aquí)» o «hacer eso no cambiará mucho las cosas», pero muchos equipos se dan cuenta de que es más fácil de implementar de lo que parece y pasado un tiempo su reacción cambia a «¿cómo puedes vivir sin eso?». Ahora es tu elección si decides probarlo o no, pero antes de hacerlo piensa en lo poco que tienes que perder y lo mucho que puedes ganar.

Desarrollo dirigido por pruebas

Joaquín Caraballo

El desarrollo dirigido por pruebas (o TDD por sus siglas en inglés) consiste en escribir la prueba que demuestra una característica del software antes de la característica en sí; paso a paso, se va acrecentando una batería de pruebas con el código para explotación, al ritmo de:

> rojo - verde - refactorización

Donde en el paso rojo escribimos una prueba que el software no satisface, en el verde modificamos el código de explotación para que la satisfaga y en el de refactorización mejoramos el código sin cambiar funcionalidad, con la confianza de que la batería de pruebas que hemos ido creando hasta el momento nos protege de estropear algo sin darnos cuenta.

Ejemplo

Probablemente la forma más fácil de explicar TDD es con un ejemplo a nivel unitario. Supongamos que nuestra aplicación necesita convertir millas a kilómetros; empezamos con una prueba:

Rojo

```
test("Converts miles into kilometers") {
  MilesToKilometersConverter.convert(1.0) should be(1.609)
}
```

Primera prueba (example0/MilesToKilometersConverterTest.scala)

A continuación, como estamos usando para los ejemplos Scala, que es un lenguaje compilado, antes de poder ejecutar las pruebas necesitamos escribir un poco más de código. Nos limitaremos a añadir el mínimo código de explotación necesario para que compile. [8]

```
object MilesToKilometersConverter {
  def convert(miles: Double): Double =
    throw new RuntimeException("Not yet implemented")
}
```

Código necesario para que la primera prueba compile

Para terminar el paso rojo, ejecutamos la prueba, comprobando que falla (lo que muchos entornos marcan en rojo) lanzando la excepción que esperábamos:


```
Not yet implemented
java.lang.RuntimeException: Not yet implemented
```

Aunque este paso parezca trivial, nos ha obligado a tomar unas cuantas decisiones importantes:

Primero, nos ha obligado a definir el contrato de uso del conversor; en este caso, el nombre sugiere que convierte específicamente millas a kilómetros, y no a la inversa; el método de conversión forma parte del objeto (y no de la clase) `MilesToKilometersConverter`, por lo que no es necesario crear un objeto conversor llamando a `new`, y sugiere que no tiene estado [9]; las millas están expresadas con tipo `Double`; etc.

Segundo y más fundamental, la prueba constituye un ejemplo que comienza a definir la funcionalidad de la unidad; que al estar expresado en un lenguaje de programación nos protege de las ambigüedades del lenguaje natural. Por ejemplo, no dudaremos de la definición de milla:

> Cuando dijimos millas, ¿eran millas internacionales (1,609km)? ¿Millas náuticas (1,852km)? ¿Millas nórdicas (10km)?

Porque la prueba ha dejado claro que el tipo de milla que importa en nuestra aplicación *es igual a 1,609 kilómetros*.

Aunque a veces parezca innecesario, merece la pena, antes de progresar al paso verde, completar el rojo llegando a ejecutar una prueba que *sabemos* que va a fallar; esto nos obliga a pensar en la interfaz que estamos creando, y con frecuencia desvela suposiciones erróneas... especialmente cuando resulta que la prueba no falla, o que lo hace de una forma distinta de la que esperábamos.

Verde

A continuación acrecentamos el código de explotación. Lo más purista es escribir sólo el código imprescindible para superar la prueba:

```
object MilesToKilometersConverter {
  def convert(miles: Double): Double = 1.609
}
```

Código mínimo para que la prueba pase

Y ejecutar la prueba, que nuestra implementación superará con un color verde, si usamos un entorno que lo represente así.

¿Refactorización?

En esta iteración es pronto para refactorizar. Pasamos al siguiente paso.

Rojo

Antes de saltar a la implementación general, merece la pena que consideremos otras condiciones de entrada: ¿aceptamos distancias negativas? Como en este caso sí las aceptamos, lo expresamos con otra prueba.

```
test("Converts negative miles into kilometers") {
  MilesToKilometersConverter.convert(-2.0) should be(-3.218)
}
```

Segunda prueba (example0/MilesToKilometersConverterTest.scala)

Ejecutamos las pruebas para ver como la nueva falla.

```
1.609 was not equal to -3.218
org.scalatest.TestFailedException: 1.609 was not equal to -3.218
```

Verde

Ahora que tenemos dos ejemplos de la funcionalidad de `convert`, es un buen momento para buscar una implementación más general [10] que no sólo satisfaga estas dos pruebas, sino que también nos proporcione la funcionalidad general de la que son ejemplo. Como la siguiente:

```
object MilesToKilometersConverter {
  def convert(miles: Double): Double = miles * 1.609
}
```

Código para la segunda prueba (example0/MilesToKilometersConverter.scala)

Podríamos haber pasado a la solución general directamente, y a menudo haremos exactamente eso, sin embargo, como vimos en el anterior paso verde, empezar por la solución más simple y dejar la generalización para el siguiente ciclo, nos ayuda a centrarnos primero en definir qué funcionalidad vamos a añadir y cuál va a ser el contrato de la unidad que estamos probando, resolviendo ambigüedades y desvelando suposiciones erróneas. Esto será particularmente útil cuando no tengamos claro lo que queremos hacer y cómo queremos implementarlo.

Refactorización

Tras cada paso verde debemos plantearnos la mejora del código, con la confianza de que las pruebas nos van a proteger de romper algo que funcionaba anteriormente. Las refactorizaciones pueden centrarse en el detalle (¿es mejor expresar `miles * 1.609` o `1.609 * miles`? ¿el parámetro debería llamarse `miles` o `distanceInMiles`?), pero es fundamental que con frecuencia también reconsideremos el diseño de la aplicación y lo transformemos en lo más apropiado para el estado actual del código, sin ignorar la dirección futura en la que queremos ir pero sin fraguar demasiado lo que puede que nunca necesitemos.

Por qué y para qué

El motivo más importante [11] para desarrollar dirigido por pruebas es el énfasis en la función de nuestro código y en su idoneidad para ponerlo a prueba y por lo tanto usarlo. Este énfasis nos obliga a preguntarnos cada vez que vamos a añadir código de explotación: ¿es esto lo que necesito? ¿hace falta *ahora*?; ayudándonos a escribir exclusivamente lo que necesitamos y a mantener el tamaño y la complejidad del código al mínimo necesario.

Si bien TDD no exime de buscar y dirigir el diseño deliberadamente, escribir código que, desde el primer momento, es fácil de probar favorece una cierta simplicidad y, definitivamente, evidencia el acoplamiento, guiándonos hacia el cuidado de la colaboración entre las unidades. En particular, la inyección de dependencias y la separación entre sus interfaces y las implementaciones, emergen de forma natural, dado que facilitan las pruebas automatizadas.

Los proyectos que se desarrollan dirigidos por pruebas cuentan en todo momento con una batería de pruebas al día, que documenta la intención de cada unidad del software, de combinaciones de unidades y del software en su totalidad. Además, las pruebas, si bien no la garantizan, dan una buena indicación de la corrección del software; lo que reduce el miedo a romper algo, y lo sustituye por un hábito diligente de refactorizar con frecuencia y mejorar el diseño progresivamente.

Ejemplo de cómo las pruebas nos guían respecto a la cohesión y el acoplamiento

En la clase siguiente, el método `translate` traduce un texto del español a una especie de inglés, mostrando por pantalla el resultado; este código es poco probable que haya sido desarrollado guiado por pruebas, dado que el resultado principal, la traducción, no se pone a disposición del codigo externo de ninguna manera que sea fácil de incluir en una prueba, sino que se manifiesta llamando al método `println` que da acceso a la consola, es decir, mediante un *efecto secundario*, lo que dificulta la verificación desde una prueba.

```
class SpanishIntoEnglishTranslator {
  def translate(spanish: String) {
    println(spanish.split(' ').map(_ match {
      case "yo" => "I"
      case "soy" => "am"
      case _ => "mmmeh"
    }).mkString(" "))
  }
}
```

Código difícil de probar (example1coupled/SpanishIntoEnglishTranslator.scala)

Si lo desarrollamos con la facilidad de prueba en mente desde el principio, probablemente nos encontraremos con que, para probar el resultado de la traducción, necesitamos que el código que traduce devuelva el resultado; de hecho, ¿acaso no es la traducción en sí la responsabilidad principal de esta clase, y no el mostrar por pantalla? Si pudiéramos obtener el resultado, una prueba de nuestro traductor podría ser algo así:

```
var translator: SpanishIntoEnglishTranslator = _

before {
  translator = new SpanishIntoEnglishTranslator()
}

test("translates what it can") {
  translator.translate("yo soy") should be("I am")
}

test("mmmehs what it can't") {
  translator.translate("dame argo") should be("mmmeh mmmeh")
}
```

Primera prueba para el traductor (example2/SpanishIntoEnglishTranslatorTest.scala)

Lo que nos llevaría a un traductor menos acoplado a la muestra por pantalla

```
class SpanishIntoEnglishTranslator {
  def translate(spanish: String): String =
    spanish.split(' ').map(_ match {
      case "yo" => "I"
      case "soy" => "am"
      case _ => "mmmeh"
    }).mkString(" ")
}
```

Un traductor más fácil de probar (example2/SpanishIntoEnglishTranslator.scala)

Probar el conjunto de la aplicación

Hasta ahora nos hemos centrado en las pruebas unitarias; no obstante, si somos consecuentes con los principios que hemos visto —guiarnos manteniendo el enfoque en los objetivos del software, documentar y verificar—, deberemos considerar fundamental guiar el desarrollo de cada parte de la funcionalidad mediante una prueba que la ejercite en su conjunto; lo ideal será que todas las pruebas funcionales verifiquen el conjunto del software, en un entorno similar al de explotación, o incluso en el entorno de explotación en sí. En la práctica suele haber muchos obstáculos, por ejemplo, puede que sea demasiado costoso llevar a cabo *de verdad* ciertas acciones destructivas, que no haya suficientes recursos, o que se hayan impuesto decisiones arquitectónicas que dificulten las pruebas; sin embargo, eso no significa que tengamos que claudicar completamente; a menudo, lograremos las mejoras más importantes en el software y en la organización en la que se crea cuestionando las limitaciones.

Ejemplo

Volvamos al ejemplo inicial de conversión de distancias, y supongamos que necesitamos ofrecer a nuestros clientes un servicio de conversión de unidades a través de un servicio web, porque hemos decidido que no hay suficientes conversores en Internet. La *primera* prueba que vamos a escribir, incluso antes de las que vimos en la introducción, es una prueba que ejercite el conjunto de la aplicación. Nos concentraremos en un cierto mínimo incremento de funcionalidad, visible para los usuarios del sistema, que requiera una implementación reducida y que tenga un alto valor desde un punto de vista comercial o de los objetivos últimos del proyecto. En nuestro caso empezamos con la conversión de millas a kilómetros.


```
class ConversionTest
    extends FunSuite with ShouldMatchers with BeforeAndAfter {

  test("Converts miles into kilometers") {
    get("http://localhost:8080/1.0") should be("1.609")
  }

  def get(url: String): String = {
    Request.Get(url).execute().returnContent().asString()
  }

  var converter: Server = _

  before {
    converter = Converter.start(8080)
  }

  after {
    converter.stop()
  }
}
```

Prueba funcional para el conversor de millas (step1/functional/ConversionTest.scala)

El método `get` es aquí un método de ayuda para pruebas, que hace una petición *HTTP get* y devuelve en contenido del cuerpo del mensaje. Evidentemente, poner esto en funcionamiento requiere un cierto trabajo, pero si nos concentramos en lo fundamental, no será tanto y además nos ayudará a plantearnos cuestiones importantes acerca del sistema, particularmente a nivel de aplicación, por ejemplo: *¿cómo nos comunicaremos con el sistema?*; y acerca de cómo lo vamos a probar. Así, desde el primer momento la facilidad de prueba es un *usuario de pleno derecho* de nuestro proyecto.

Con esta prueba como guía, nos concentraremos ahora en recorrer todo el sistema, casi a toda velocidad, hasta que la satisfagamos. En el mundo de Java/Scala, la forma típica de resolver esto es con un Servlet. De nuevo comenzamos con una prueba, esta vez a nivel unitario. [12]

```
class ConverterTest extends FunSuite {
  test("Responds to get requests converting miles into kilometers") {
```

```
    val response = mock(classOf[HttpServletResponse])
    val printWriter = mock(classOf[PrintWriter])
    when(response.getWriter).thenReturn(printWriter)

    new Converter().doGet(mock(classOf[HttpServletRequest]), response)

    verify(printWriter).print("1.609")
  }
}
```

Prueba unitaria para el conversor de millas (step1/ConverterTest.scala)

Un conversor más o menos mínimo usando Jetty viene a ser:

```
class Converter extends HttpServlet {
  override def doGet(req: HttpServletRequest, resp: HttpServletResponse) {
    resp.getWriter.print("1.609")
  }
}

object Converter {
  def main(args: Array[String]){
    start(8080)
  }

  def start(port: Int): Server = {
    val context = new ServletContextHandler()
    context.setContextPath("/")
    context.addServlet(new ServletHolder(new Converter()), "/*")

    val converter = new Server(port)
    converter.setHandler(context)

    converter.start()
    converter
  }
}
```

Primera implementación del servicio de conversión de millas (step1/Converter.scala)

Como vemos la funcionalidad que estamos ofreciendo es, como en el ejemplo inicial, trivial. Pero llegar a ella nos ha obligado a definir el esqueleto de todo nuestro sistema, incluyendo código de explotación y de prueba.

A continuación progresaremos dependiendo de nuestras prioridades. Por ejemplo, podemos concentrarnos en completar funcionalmente la conversión de millas a kilómetros.


```
test("Converts negative miles into kilometers") {
  get("http://localhost:8080/-2.0") should be("-3.218")
}
```

Segunda prueba funcional del conversor de millas (step2/functional/ConversionTest.scala)

```
class Converter extends HttpServlet {
  override def doGet(req: HttpServletRequest, resp: HttpServletResponse) {
    val miles = req.getRequestURI.substring(1).toDouble
    resp.getWriter.print(miles * 1.609)
  }
}
```

Código para la segunda prueba funcional del conversor de millas (step2/Converter.scala)

A continuación el manejo de los casos de error, como cantidades de millas que no sean numéricas

```
test("Responds with 400 (Bad Request) and error message to unparseable amounts of miles") {
  statusCode("http://localhost:8080/blah") should be(400)
  get("http://localhost:8080/blah") should be("Miles incorrectly specified: /blah")
}
```

Prueba para caso de error del conversor de millas (step3/functional/ConversionTest.scala)

```
class Converter extends HttpServlet {
  override def doGet(req: HttpServletRequest, resp: HttpServletResponse) {
    val milesAsString = req.getRequestURI.substring(1)
    try {
      val miles = milesAsString.toDouble
      resp.getWriter.print(miles * 1.609)
    }
    catch {
      case _: NumberFormatException => {
        resp.setStatus(HttpServletResponse.SC_BAD_REQUEST)
        resp.getWriter.print("Miles incorrectly specified: " + req.getRequestURI)
      }
    }
  }
}
```

Código de manejo de errores para el conversor de millas (step3/Converter.scala)

Además de los pasos *rojos* y *verdes* que hemos visto en el ejemplo hasta ahora, a medida que la aplicación va creciendo, debemos tanto refactorizar a nivel unitario como ir mejorando el diseño; por ejemplo, si en los próximos pasos la aplicación necesitase

responder a distintas rutas con distintas conversiones, probablemente decidiríamos extraer el análisis de las URIs a una unidad independiente e introduciríamos distintos objetos a los que delegar dependiendo del tipo de conversión.

Probar una sola cosa a la vez

El mantenimiento de la batería de pruebas, que crece con la aplicación, requiere una inversión de esfuerzo constante; hacer que cada prueba verifique únicamente un aspecto de la aplicación nos ayudará a mantener este esfuerzo manejable y además las hará más fáciles de entender, y por lo tanto más eficientes. Idealmente, el cambio de un detalle del funcionamiento de nuestra aplicación debería afectar exclusivamente a una prueba que sólo verifica ese detalle, o, dicho de otra manera:

- si es relativamente fácil cambiar un cierto aspecto del funcionamiento sin que falle ninguna prueba, tenemos una laguna en la cobertura de la batería; [13]
- si falla más de una, la batería tiene código redundante, incrementando el coste de mantenimiento;
- si la prueba que falla incluye la verificación de elementos que no están directamente relacionados con nuestro cambio, probablemente sea demasiado compleja, dado que introducir el cambio en el sistema requiere tener en cuenta aspectos independientes de la aplicación.

Ejemplo de pruebas centradas

Volvamos a donde dejamos el ejemplo del traductor y supongamos que lo siguiente que queremos hacer es separar las palabras del texto original no sólo mediante espacios, sino también mediante cambios de línea. Como estamos guiando los cambios con pruebas, añadimos a `SpanishIntoEnglishTranslatorTest` una prueba que verifique el nuevo funcionamiento.

```
test("splits by change of line") {
  translator.translate("yo\nsoy") should be("I am")
}
```

Ejemplo de prueba no centrada

El problema que tiene esto es que la prueba mezcla la separación del texto original y la traducción de las palabras; la idea que queremos transmitir con este ejemplo estaría más clara si pudiéramos expresar la entrada como `"xxx\nxx"` y la condición a cumplir como `should be(Seq("xxx", "xx"))`; sin embargo, la forma actual del sistema no lo permite, porque la traducción es parte del método que estamos probando.

Supongamos además que el siguiente incremento funcional afectase a la traducción de palabras en sí, por ejemplo, cambiando el idioma origen al francés o a otra variante del español; este cambio afectaría a cada una de las pruebas de `SpanishIntoEnglishTranslatorTest`, pero, ¿por qué debería verse afectada una prueba como `test("splits by change of line")`, cuyo propósito es probar la separación en palabras?

Podemos ver estas deficiencias de nuestra batería como el resultado de una granularidad inapropiada, dado que la misma prueba está verificando varias cosas: separación, traducción y reunión de palabras. La solución consistiría en refactorizar antes de aplicar el cambio: ¿Quizá la clase que se encarga de descomponer y componer debería ser distinta de la que traduzca palabra por palabra?

Extraemos la división de palabras a su propia unidad, con lo que podemos expresar, con una prueba más centrada, la división del texto a traducir por saltos de línea:

```
class SplitterTest extends FunSuite with ShouldMatchers {
  test("splits by space") {
    Splitter("xxx xx") should be(Seq("xxx", "xx"))
  }

  test("splits by change of line") {
    Splitter("xxx\nxx") should be(Seq("xxx", "xx"))
  }
}
```

Prueba del divisor de palabras (example3/SplitterTest.scala)

```
object Splitter extends ((String) => Seq[String]) {
  def apply(s: String): Seq[String] = s split """[ \n]"""
}
```

Código para el divisor de palabras (example3/Splitter.scala)

El traductor lo recibe como una dependencia inyectada mediante el constructor. [14]

```
class SpanishIntoEnglishTranslatorTest
    extends FunSuite with ShouldMatchers with BeforeAndAfter {

  var translator: SpanishIntoEnglishTranslator = _

  before {
    translator = new SpanishIntoEnglishTranslator(_ split ' ')
  }

  test("translates what it can") {
    translator.translate("yo soy") should be("I am")
  }

  test("mmmehs what it can't") {
    translator.translate("dame argo") should be("mmmeh mmmeh")
  }
}
```

Pruebas para un traductor desacoplado del divisor de palabras (example3/SpanishIntoEnglishTranslatorTest.scala)

```
class SpanishIntoEnglishTranslator(val splitter: (String) => Seq[String]) {
  def translate(spanish: String): String = {
    val split = splitter(spanish)
    split.map(_ match {
      case "yo" => "I"
      case "soy" => "am"
      case _ => "mmmeh"
    }).mkString(" ")
  }
}
```

Código del traductor desacoplado del divisor de palabras (example3/SpanishIntoEnglishTranslator.scala)

El aumento de la granularidad nos ha permitido que la introducción de funcionalidad nueva no afecte a multitud de pruebas. Sin embargo, esto no ha sido gratis; hemos aumentado la complejidad del código. Al final, todas estas decisiones hay que valorarlas una a una y decidir qué es lo más apropiado en cada caso, teniendo en cuenta aspectos como la complejidad, el tiempo de ejecución y la dirección en la que esperamos y queremos que vaya el proyecto.

Mantener verde la batería

La batería de pruebas es la documentación de la funcionalidad de nuestro código. Una documentación que se mantiene al día, porque va creciendo con cada cambio y es ejercitada, es decir, ejecutamos las pruebas, al menos con cada envío de los cambios al repositorio.

Trabajar dirigido por pruebas significa mantener siempre el correcto funcionamiento del sistema; idealmente la última versión en el repositorio común deberá estar en todo momento lista para ponerla en explotación, y las pruebas satisfechas en todo momento [15], con lo que la documentación proporcionada por las pruebas estará siempre al día. Para lograrlo, deberemos comprobar la satisfacción de las pruebas antes de enviar cualquier cambio al repositorio común; además, muchos equipos se ayudan de un sistema de integración continua que verifica automáticamente la batería cada vez que se detecta un cambio en el repositorio.

A medida que crece la aplicación, el tiempo que requiere la batería completa tiende a aumentar, lo que incrementa el coste del desarrollo y motiva a los desarrolladores a no siempre satisfacer la batería completa o a espaciar los envíos de cambios al repositorio común; ambos efectos muy perniciosos. Para mantener viable la programación dirigida por pruebas, debemos esforzarnos en mantener reducido este tiempo; la manera de lograrlo va más allá de un artículo introductorio, pero incluye la selección y el ajuste de la tecnología empleada para los distintos elementos de la batería, la ejecución en paralelo e incluso la partición de la aplicación en sí o cualquier ajuste que la haga más rápida.[16]

Otro problema relacionado con el coste de la batería son los fallos intermitentes, que necesitarán un esfuerzo importante de mantenimiento; hemos de invertir el esfuerzo necesario para entender la raíz de cada fallo y resolverlo. Las fuentes típicas de fallos intermitentes son los aspectos no deterministas del software; por ejemplo, cuando lo que verificamos es de naturaleza asíncrona, necesitamos controlar el estado de la aplicación mediante puntos de sincronización. Algunas condiciones son imposibles de verificar per se, como la ausencia de comportamiento; en estos casos a menudo la solución pasa por alterar la aplicación desvelando su estado lo suficiente como para que las pruebas puedan sincronizarse con la aplicación y sepan cuándo debería haber tenido lugar el evento que estemos verificando. También suelen causar fallos intermitentes las dependencias de elementos fuera del control de la prueba, como el reloj del sistema; y su solución pasa normalmente por la inclusión y el control de dicho elemento desde la prueba, por ejemplo, alterando la percepción del tiempo de la aplicación mediante un *proxy* controlable desde las pruebas.

Críticas

Diseño prematuro

El ejemplo de pruebas centradas ilustra también una de las principales críticas contra el desarrollo dirigido por pruebas: para poder probar la clase de traducción satisfactoriamente, la hemos descompuesto en un diccionario y un desensamblador/ensamblador de palabras; sin embargo, si de verdad fuéramos a diseñar un sistema de traducción automatizada esta abstracción no sería apropiada, ya que el diccionario necesita el contexto, la disposición de palabras en el texto resultante depende de la función gramatical, etc. ¿Significa esto que el TDD nos ha guiado en la dirección incorrecta?

Como dijimos antes, desarrollar dirigidos por pruebas significa considerar las pruebas como usuarios de pleno derecho de nuestro código, añadiendo, así, un coste en cantidad de código y en la complejidad del diseño que elegimos pagar para beneficiarnos de la guía de las pruebas; optar por el TDD es considerar que este coste vale la pena. Sería menos costoso desarrollar sin la guía de las pruebas si supiésemos exactamente cuáles son los requisitos e incluso qué código escribir desde el principio. El TDD se opone a esta visión considerando el desarrollo como un proceso progresivo en el que el equipo va descubriendo *qué* y *cómo* desarrollar, a medida que crea la aplicación y la va poco a poco transformando, enriqueciendo y simplificando, pasando siempre de un sistema que funciona a un otro sistema que funciona, y que hace quizá un poco más que el anterior.

Volviendo a la prematuridad del diseño, si bien es cierto que las pruebas a veces adelantan la necesidad de descomponer el código, estas decisiones se vuelven más baratas dado que los cambios en el diseños son menos costosos gracias a la protección que nos da nuestra batería. Y, además, esto se ve también compensado por las innumerables ocasiones en las que no añadiremos complejidad al código de explotación porque esa complejidad no es necesaria para satisfacer las pruebas, probablemente porque no lo es en absoluto para lo que requerimos de nuestra aplicación en ese momento.

Guiar sólo mediante pruebas funcionales

Algunos equipos experimentados deciden utilizar las pruebas para guiar el desarrollo a nivel funcional, pero no a nivel unitario, sólo escribiendo pruebas unitarias cuando no resulta práctico cubrir con pruebas funcionales ciertas partes del código. Esto lo hacen porque ven las pruebas unitarias más como un lastre que como una guía útil en el diseño interno de la aplicación, quizá porque consideran que su criterio es suficiente para lograr un buen diseño interno. Nuestra recomendación es comenzar guiando todas las unidades por pruebas y progresar hacia un equilibrio en el que no probemos a nivel unitario el código que sea obvio, pero donde reconozcamos que el código que no merece la pena guiar mediante pruebas por ser obvio probablemente es poco útil [17], y que el que es difícil de probar probablemente peca de acoplado. La recomendación es, en definitiva, aplicar el sentido común pero no renunciar a la guía que nos proporcionan las pruebas en el desarrollo de los niveles inferiores de la aplicación.

Conclusión

El desarrollo dirigido por pruebas nos ayuda a construir aplicaciones donde aquello que nos motiva a escribir el software guía cada línea de código, evitando a cada nivel desperdiciar nuestros esfuerzos en desarrollar funcionalidad que no sea exactamente la que necesitamos, o que sea simplemente innecesaria. La guía de las pruebas, combinada con el diseño continuo, hace posible un estilo de desarrollo orgánico, en el que el equipo evoluciona la aplicación, a medida que mejora su conocimiento de los requisitos del dominio y de la implementación ideal para resolverlos. Además, la utilización de pruebas desde el primer momento, desvela el acoplamiento y nos incita a descomponer el código en unidades cohesivas con depencias claramente identificadas.

Si bien no hay un concenso absoluto en la idoneidad del desarrollo guiado por pruebas, muchos equipos lo usan, particularmente para aplicaciones comerciales y en lenguajes orientados a objetos. Nuestra opinión es que, además de dar sentido y ritmo a cada actividad del desarrollo, lo hace más divertido, porque da al desarrollador más libertad para mejorar y acrecentar el software.

Bibliografía

Lecciones de aprender un lenguaje funcional

- [onlisp] Paul Graham. «On Lisp». Prentice Hall. ISBN 0130305529. http://www.paulgraham.com/onlisp.html
- [landoflisp] Conrad Barski. «Land of Lisp». No Starch Press. ISBN 978-1-59327-281-4. http://landoflisp.com/
- [learnhaskell] Miran Lipovača. «Learn You a Haskell for Great Good!». No Starch Press. ISBN 978-1-59327-283-8. http://learnyouahaskell.com/
- [progscala] Dean Wampler y Alex Payne. «Programming Scala». O'Reilly Media. ISBN 978-0-596-15595-7. http://ofps.oreilly.com/titles/9780596155957/
- [proginscala] Martin Odersky, Lex Spoon, y Bill Venners. «Programming in Scala». Artima. ISBN 9780981531601. http://www.artima.com/pins1ed/
- [jsfuncional] Dmitry A. Soshnikov. «JavaScript array "extras" in detail». http://dev.opera.com/articles/view/javascript-array-extras-in-detail/

Documentación activa

- [goos] Steve Freeman y Nat Price. «Growing Object-Oriented Software Guided by Tests». Addison-Wesley Professional. ISBN 978-0321503626. http://www.growing-object-oriented-software.com/
- [concordion] David Peterson. «Concordion». http://concordion.org/
- [xcordion] Robert Pelkey. «XCordion» (clon de Concordion). http://code.google.com/p/xcordion/
- [bridging] Gojko Adzic. «Bridging the Communication Gap». Neuri Limited. 978-0955683619. http://www.acceptancetesting.info/the-book/

Siete problemas al probar programas

- [pseudocode] Esteban Manchado Velázquez. «From pseudo-code to code». http://hcoder.org/2010/08/10/from-pseudo-code-to-code/
- [testing123] Esteban Manchado Velázquez. «Software automated testing 123». http://www.demiurgo.org/charlas/testing-123/
- [fowlerci] Martin Fowler. «Continuous Integration». http://www.martinfowler.com/articles/continuousIntegration.html
- [kivaapi] API pública de Kiva. http://build.kiva.org/docs/
- [loanmeter] Esteban Manchado Velázquez. «World Loanmeter». https://github.com/emanchado/world-loanmeter
- [jasmine] Página web de Jasmine, un módulo de pruebas para Javascript. http://pivotal.github.com/jasmine/
- [pydoubles] PyDoubles Test doubles framework. http://www.pydoubles.org/

Calidad en software

- [technopoly] Neil Postman. «Technopoly». Random House USA Inc. ISBN 9780679745402.
- [theletter] Uncle Bob. «The Letter». http://blog.8thlight.com/uncle-bob/2012/01/12/The-Letter.html
- [obliquestrategies] Brian Eno y Peter Schmidt. «Oblique Strategies». http://en.wikipedia.org/wiki/Oblique_Strategies
- [broken] Seth Godin. «This is broken» (TED Talk). http://www.ted.com/talks/seth_godin_this_is_broken_1.html
- [livingcomplexity] Donald A. Norman. «Living with Complexity». The MIT Press. ISBN 9780262014861.

Integración continua

- [continuosintegration] Paul M Duvall. «Continuos Integration: Improving Software Quality and Reducing Risk». Addison-Wesley Professional. ISBN 978-0321336385
- [continuosdelivery] Jez Humble y David Farley. «Continuos Delivery». Addison-Wesley Professional. ISBN 978-0321601919
- [fowlerci-bis] Martin Fowler. «Continuous Integration». http://www.martinfowler.com/articles/continuousIntegration.html
- [xpexplained] Kent Beck y Cynthia Andres. «Extreme Programming Explained: Embrace Change». Addison-Wesley Professional. ISBN 978-0321278654
- [bleyco] Carlos Blé Jurado. «Diseño ágil con TDD». ISBN 978-1445264714. http://dirigidoportests.com/el-libro
- [xunit] Wikipedia. «Frameworks xUnit». http://en.wikipedia.org/wiki/XUnit

Desarrollo dirigido por pruebas

- [tddbyexample] Kent Beck. «Test Driven Development: By example». Addison Wesley. ISBN 978-0321146533.

- [tddequality] Kent Beck. «Test Driven Development: Equality for All». InformIT. http://www.informit.com/articles/article.aspx?p=30641
- [goos-bis] Steve Freeman y Nat Pryce. «Growing Object-Oriented Software Guided by Tests». Addison-Wesley Professional. ISBN 978-0321503626.
- [mockito] Szczepan Faber and friends. «Mockito: simpler & better mocking» (sitio del proyecto). http://code.google.com/p/mockito/
- [bleyco-bis] Carlos Blé Jurado. «Diseño ágil con TDD». ISBN 978-1445264714. http://dirigidoportests.com/el-libro

---

1. Las pruebas de integración son las más completas que hacemos, que determinan si el proyecto, como un todo, funciona desde el punto de vista del usuario.

2. Hacer las cosas de cierta manera simplemente porque lo hemos hecho o visto antes, sin entender por qué son así o qué utilidad tienen. Ver Cargo Cult Programming en Wikipedia.

3. Los buenos profesionales hacen las dos cosas, pero es más profesional tener más de lo segundo que más de lo primero.

4. Si siempre hacemos ciertas cosas de la misma manera y ésta funciona razonablemente bien, no tenemos que gastar tiempo ni energía decidiendo cómo hacerlas.

5. Aunque uno podría decir que «no hay ninguna regla exenta de excepciones» también tiene excepciones...

6. Reunión de aproximadamente una hora en la que se decide cuáles serán las tareas a incluir en la siguiente versión de la aplicación.

7. Breve reunión de seguimiento, diaria, que realiza todo el equipo, donde expone en qué se trabajó el día anterior, en qué se trabajará hoy y si existen impedimentos para llevar a cabo alguna de las tareas en ejecución.

8. Si estamos usando un entorno de desarrollo, la función de *arreglo* hará la mayor parte del trabajo por nosotros. En muchos lenguajes como Scala, el compilador nos obligará a incluir alguna implementación antes de permitirnos ejecutar. Algunos desarrolladores suelen implementar inicialmente los métodos lanzando una excepción como en el ejemplo, lo que ayuda a mantener la separación rojo-verde, ya que no se piensa en la implementación hasta el paso verde. Aunque esto pueda parecer prolijo, resulta bastante rápido de producir si tenemos preparada una plantilla en nuestro entorno que introducimos con un atajo. Otra opción es generar la implementación más sencilla que se nos ocurra —por ejemplo, devolviendo `0` o `null`—.

9. Para los lectores más familiarizados con Java que con Scala, los objectos creados con `object` en Scala son semejantes a la parte estática de las clases en Java; los métodos definidos bajo `object` no están ligados a ningún ejemplar (en inglés *instance*) de la clase `MilesToKilometersConverter` —que de hecho aquí no existe—. Lo cierto es que estos objetos sí que pueden tener estado (de la misma manera que podemos tener campos estáticos en Java), pero muchos equipos eligen no usar esta característica del lenguaje, a no ser que haya un buen motivo; por eso decimos que definir el método `convert` en el objeto sugiere que el conversor no tiene estado.

10. A esta generalización Kent Beck la llama *triangulación*. No estoy seguro de que me guste el término, porque la triangulación geométrica a la que hace analogía permite de forma determinista encontrar una posición a partir de los datos de que se dispone. Aquí, sin embargo, los ejemplos por sí solos no nos permitirían encontrar la solución general, que precisa que además entendamos el problema más allá de los ejemplos.

11. *Para mí* el más importante, seguro que otros discreparán.

12. En este ejemplo hemos usado dobles de prueba para verificar el comportamiento de `Converter` con sus colaboradores; simplemente estamos comprobando que, cuando llamamos al método `doGet()` pasando un objeto de tipo `HttpServletRequest` y otro de tipo `HttpServletResponse`, el método bajo prueba llama al método `getWriter` del segundo parámetro y, al devolver `getWriter` un objeto de tipo `PrintWriter`, el método bajo prueba llama al método `print` del `PrintWriter` con el parámetro `"1.609"`. Gracias a herramientas como Mockito ([mockito]), que hemos usado en el ejemplo, es más fácil explicar la interacción en código que en español. En *Diseño ágil con TDD* ([bleyco-bis]) se explica el uso de dobles y de otros aspectos de la programación dirigida por pruebas.

13. El tipo de pruebas con las que guiamos el desarrollo en este artículo documentan el comportamiento con buenos ejemplos, pero generalmente no lo garantizan para todas las posibles entradas, ni aspiran a hacerlo; por lo tanto, normalmente es muy fácil cambiar el comportamiento de forma intencionada sin que las pruebas dejen de satisfacerse —por ejemplo, con algo como `if(input==15) throw new EasterEgg`—; una buena cobertura en TDD está en el punto de equilibrio en el que sea poco probable cambiar la funcionalidad accidentalmente sin que fallen las pruebas.

14. En este ejemplo nos hemos quedado con pruebas que ejercitan las unidades unidades independientemente; perdiendo la cobertura de la combinación de ambas unidades. Sin embargo, en la práctica, este código formaría parte de un proyecto mayor que contaría con baterías de pruebas de mayor ámbito, que incluirían la verificación del comportamiento conjunto.

15. De hecho, algunos equipos hacen exactamente eso, ponen cada versión que satisface la batería completa automáticamente en explotación.

16. Otra forma de reducir el tiempo es transigiendo en alguna medida, es decir, no cumpliendo en ocasiones todo lo que describimos en este artículo.

17. En algunos casos, debido a una convención sospechosa o a las limitaciones de nuestro lenguaje de programación —por ejemplo, los métodos *set* y *get* en Java—.

Versión 2013-01-19 16:57:46 +0100 0688b6e