Capítulo 10

INTRODUCCIÓN A CUCUMBER

> El hombre es el único animal que tropieza dos veces con la misma piedra,
>
> Anónimo
>
> Los humanos son los únicos animales que repiten manualmente dos veces la misma prueba,
>
> Ángel E. García Baños

Alrededor de *Ruby* hay un colectivo enorme de excelentes programadores que han ido creando un ecosistema de herramientas siguiendo la misma filosofía que el lenguaje: fácil y potente. Entre ellas está el *framework* para programación en la web *Rails*, herramientas de *deployment* como *Capistrano* y la que vamos a presentar ahora, *Cucumber*, para hacer pruebas automáticas. Hay otras alternativas, para todos los gustos.

Habitualmente se usan los lenguajes livianos como *Ruby* bajo metodologías ágiles, concretamente BDD (*Behavior Driven Development*). Y en ellas las pruebas son algo fundamental. La recomendación es crear una batería de pruebas que debe pasar el software, y solo después escribir ese *software*. Así, de cierta manera, las pruebas se convierten en el equivalente a los requerimientos funcionales de las metodologías pesadas. De este modo, como programador puedo escribir las pruebas funcionales de las clases, las de integración, las del sistema y las de aceptación. Esto último es lo más importante: dado que las pruebas se escriben en español (o en inglés o cualquier otro idioma humano), se puede animar al cliente a que las redacte aunque no sepa nada de programación. Todas estas pruebas se convierten después a código ejecutable y un sistema de colores señalizará para cada una de ellas si

se pasó o no se pasó (rojo=no se pasó, verde=sí se pasó, amarillo=pendiente de escribirse el código de la prueba). El cliente puede saber inmediatamente si el *software* que le estamos desarrollando cumple o no con sus expectativas.

Porque sabemos que es imposible garantizar teóricamente que el *software* que escribimos esté libre de errores (a no ser que limitemos la expresividad del lenguaje de programación, para que deje de ser Turing-completo). Pero con pruebas automáticas (sea con *Cucumber* o con cualquier otra herramienta) lo que logramos es que el número de errores disminuya de forma monótona, es decir, que tienda asintóticamente a cero. Las pruebas manuales no lo garantizan y mucho menos la ausencia de pruebas de ningún tipo.

La razón es que dentro de la metodología *BDD* se especifica que cada vez que se encuentre un error en nuestro software (bien sea en el ambiente de desarrollo o en el de producción) lo primero que se debe hacer es escribir una prueba que capture el problema. Al correr todas las pruebas, esa fallará, por supuesto. Ahora es el momento de reparar el error en el programa, corriendo las pruebas hasta que todas ellas pasen de nuevo.

En ambientes de prueba manuales es habitual que, al corregir un error en una parte del programa se genere un nuevo problema en otra parte. Y que cuando se detecte y lo corrijamos, puede ocurrir que el primer error resucite. Ello ocurre porque es muy laborioso ejecutar manualmente de nuevo todas las pruebas, y lo que se suele hacer es verificar únicamente que el nuevo error reportado haya desaparecido. Mientras que en un ambiente de pruebas automático es imposible que ocurra este problema porque las pruebas quedan escritas de forma acumulativa. Se introducen nuevas pruebas, pero no se borran las anteriores. De modo que, en cada ejecución de pruebas, si un error viejo resucita nos daremos cuenta y el *software* no saldrá a producción hasta que todas las pruebas se pasen en verde.

Diseñar las pruebas con Cucumber es muy sencillo. Supongamos que nuestro objetivo es diseñar una cuenta de ahorros bancaria:

```
class Banco
  def initialize(valorInicial)
  end

  def ingresar(valor)
  end

  def retirar(valor)
  end

  def saldo
```

```
    end
end
```

Como vemos, todavía no tenemos el código, pero sí las funcionalidades que ofrecerá. Entonces se requieren dos archivos para las pruebas:

Archivo.feature. Contiene la descripción de las pruebas en español. Hay algunas pocas palabras clave que están coloreadas y que sirven para orientar cómo escribir las pruebas, pero no tienen ninguna semántica asociada, es decir, se ignoran por el *software* de pruebas. Lo que se hace es definir cada característica del *software* que se va a probar y a continuación una serie de escenarios, tanto con los casos felices como los que especifican que hacer cuando la contraseña no coincide o algún dato introducido no es correcto. A continuación vemos un ejemplo:

```
Característica: Verificar el correcto funcionamiento de una cuenta de ahorros.

  Antecedentes: Crear una cuenta de ahorros bancaria con 1000 pesos
    Dado que necesito ahorrar, abro una cuenta de ahorros con 1000 pesos iniciales

  Escenario: Verificar que la cuenta tiene el saldo de 1000 pesos
    Cuando pido el saldo
    Entonces debe decirme que tengo 1000 pesos

  Escenario: Introduzco más dinero
    Dado que ingreso 3000 pesos
    Cuando pido el saldo
    Entonces debe decirme que tengo 4000 pesos

  Escenario: Retirar dinero
    Dado que retiro 400 pesos
    Cuando pido el saldo
    Entonces debe decirme que tengo 600 pesos

  Escenario: Imposibilidad de retirar más dinero del que hay
    Dado que retiro 5000 pesos
    Cuando pido el saldo
    Entonces debe decirme que no se pudo hacer ese retiro
    Y debe decirme que tengo 1000 pesos
```

El texto posterior a la palabra Característica sirve solo para documentar y el *software* de pruebas la ignora. Lo mismo ocurre con el texto posterior a Escenario.

Todas las demás líneas de texto que comienzan por una palabra en color rojo (Dado, Dada, Dados, Dadas, Cuando, Entonces, Y, E, Pero) se convierten a un pequeño trozo de código que recrea en Ruby lo que se dice en español (enseguida explicaremos como se hace esta magia). Y los Antecedentes definen lo que se hace justo antes de que comience cada escenario. Los escenarios son independientes entre sí, es decir, los resultados de unos no afectan a los de otros. Precisamente los antecedentes sirven para dar valores iniciales y que todos los escenarios comiencen de la misma manera.

Archivo_steps.rb. Por cada archivo **feature** debe haber un archivo **steps** donde se especifique como se traduce cada sentencia en español a código Ruby. Es aquí donde se hace la magia, que consiste en que no hay ninguna magia, pues el programador debe hacer la traducción manualmente. Por ejemplo, para el caso anterior sería así:

```
Dado /^que necesito ahorrar, abro una cuenta de ahorros con 1000 pesos
iniciales$/ do |valorInicial|
  @banco = Banco.new(valorInicial.to_i)
end

Cuando /^pido el saldo$/
  @saldo = @banco.saldo
end

Entonces /^debe decirme que tengo (.+?) pesos$/  do |nuevoSaldo|
  expect(@saldo).to eq(nuevoSaldo.to_i)
end

Dado /^que ingreso (.+?) pesos$/  do |valor|
  @banco.ingreso(valor.to_i)
end

Dado /^que retiro (.+?) pesos$/  do |valor|
  @banco.retiro(valor.to_i)
end
```

Esto que estamos viendo son algo así como funciones en Ruby, excepto que en vez de tener un nombre de función tienen una expresión regular, que incluso puede capturar datos variables. Si son numéricos, hay que conver-

tirlos de *string* a entero con la función **to_i**. La idea es que el cuerpo de la función se ejecuta cada vez que hay una coincidencia de la expresión regular con las líneas que especifican lo que hay que hacer en cada escenario del archivo **feature**. Estas funciones hacen llamadas al **Banco**, para crearlo, ingresar dinero y mirar el saldo. Pero también usan la librería **RSpec** (Chelimsky, 2010) para verificar expectativas. Allí podemos ver cómo se comparan dos números enteros por medio de la función **expect**. Si la expectativa se cumple, aparecerá la prueba en color verde, que significa que se pasó correctamente. Y si no se cumple, aparecerá de color rojo, que significa que nos hemos equivocado en algo y hay que revisar el código.

Inicialmente todas las pruebas saldrán en amarillo (no implementadas) o en rojo (fallidas) y ahora lo que toca hacer es añadir el código correcto a la clase Banco para que vaya pasando cada una de las pruebas.

Lo bonito de los pasos (**steps**) es que son reutilizables. Y así vemos que en el archivo *features* podemos usar el mismo paso varias veces con diferentes valores, como en:

Dado que retiro 400 pesos
Dado que retiro 5000 pesos

En ambos casos se ejecutará el mismo paso en Ruby (la función disparada por la expresión regular:

/^que retiro (.+?) pesos$/

Pero en cada caso la expresión regular capturará un valor distinto (400 o 5000).

Hay algunas cosas más que saber para usar *Cucumber*, como que puede haber muchos archivos *feature* y *steps* y que se pueden especificar las características con tablas, para no hacer repetitivas las sentencias en español, etc.

También conviene saber que *Cucumber* se puede conectar a muchas otras herramientas. Una de las más interesantes que he encontrado es *Selenium* (Se, 2017), que permite hacer *scripts* que naveguen solos por la web. Verás el navegador dar saltos él solito de una página a otra, rellenar campos y todo lo que un humano pueda hacer. No es la única herramienta para probar páginas web (hay otras como *watir*), pero sí es la más completa y profesional. Además, no solo sirve para hacer *testing* de las interfaces de usuario web, sino que también puedes automatizar procesos en páginas web que no ofrezcan ninguna otra alternativa como Corba o *Web Services*. Usa directamente la página web sobre los navegadores *Chromium* o *Mozilla*, imitando los clic de mouse que pueda hacer un usuario, rellenando formularios de forma programática y leyendo los resultados que entregue la página web. Al interactuar con el navegador, ejecuta cualquier *script* de *Javascript* que pudiera haber allí (esa es una ventaja respecto a *Watir*, que se limita a inte-

ractuar con la aplicación web usando mensajes HTML, sin levantar primero un navegador, con lo que se pierden todas las funcionalidades de los *scripts* del lado del cliente).

Para saber más

- David Chelimsky, Dave Astels, Zach Dennis, *et ál*. (2010). *The RSpec Book. Behaviour-Driven Development with RSpec, Cucumber, and Friends*. The Pragmatic Bookself.

Cucumber puede conectarse a muchas otras herramientas. Con *RSpec* se pueden hacer aserciones sobre condiciones que debe cumplir alguna variable del programa.

- Matt Wynne y Aslak Hellesøy (2012). *The Cucumber Book. Behaviour-Driven Development for Testers and Developers*. The Pragmatic Bookself.

Este es el mejor libro que conozco sobre Cucumber. Viene todo muy bien explicado. Introduce progresivamente los temas, primero lo más sencillo. Y tiene al menos un ejemplo de cada tema.

Referencias

Libros, artículos y enlaces web

Dees, I., Wynne, M. y Hellesøy, A. (2013). *Cucumber recipes. Automate anything with BDD tools and techniques*. Dallas: The Pragmatic Bookself.

Se (2017). SeleniumHQ Browser Automation. Recuperado el 14 de septiembre de 2017. Disponible en: https://goo.gl/kb7Xdw

Ye, W. (2013). *Cucumber BDD. How-to*. Birmingham: Packt Publising.