MÁSTER EN INVESTIGACIÓN EN INGENIERÍA DEL SOFTWARE Y SISTEMAS INFORMÁTICOS

Trabajo Fin de Máster

Itenerario Ingeniería del Software (Código: 31105151)

DSL DE PRUEBAS UNITARIAS EN RUBY

Alumno: Jesús Pardo Sánchez

Director: Ismael Abad Cardiel

Curso Académico 2016/2017

Convocatoria Junio

MÁSTER EN INVESTIGACIÓN EN INGENIERÍA DEL SOFTWARE Y SISTEMAS INFORMÁTICOS

ITINERARIO: Ingeniería del Software

CÓDIGO ASIGNATURA: 31105151

TÍTULO DEL TRABAJO: DSL de Pruebas Unitarias en Ruby

TIPO DE TRABAJO: TIPO B, propuesto por el alumno.

ALUMNO: Jesús Pardo Sánchez

DIRECTOR: Ismael Abad Cardiel

HOJA DE CALIFICACIONES

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 08…/…06/…2017….

Quién suscribe:

Autor(a): Jesús Pardo Sánchez

D.N.I/N.I.E/Pasaporte.: 44383379A

Hace constar que es la autor(a) del trabajo:

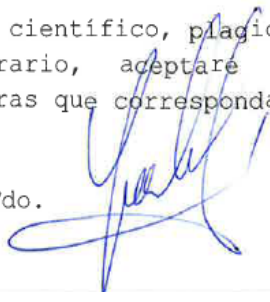
DSL de Pruebas Unitarios en Ruby

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.



Hoja Autorización

IMPRESO TFDM05_AUTOR
AUTORIZACIÓN DE PUBLICACIÓN
CON FINES ACADÉMICOS

UNED ETS de Ingeniería Informática

Impreso TFdM05_Autor. Autorización de publicación

y difusión del TFdM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

RESUMEN

Una prueba unitaria es una manera de comprobar el correcto funcionamiento de un módulo de código. El objetivo de las pruebas unitarias es aislar cada parte del programa y mostrar que las partes individuales son correctas, así como detectar errores de entendimiento en etapas tempranas del desarrollo software. Por ello, es importante que las pruebas unitarias realizadas sean comprensibles y legibles por las partes involucradas en el desarrollo software (ej. programadores, analistas, probadores, etc.). La definición de las pruebas consiste en la codificación del código fuente necesario para la ejecución automática de dichas pruebas unitarias, utilizando editores de texto plano o editores de código fuente para ello. Dada la relevancia de las pruebas unitarias existe un amplio número de marcos de trabajo para realizar pruebas unitarias en cada uno de los diferentes lenguajes de programación, dirigidas principalmente a desarrolladores de código.

Esta investigación propone un modelo para la realización de pruebas unitarias en Ruby basado en un lenguaje específico de dominio interno, inspirado en la idea de usar el lenguaje natural para expresar escenarios de pruebas unitarias, que permite identificar claramente el objetivo de las pruebas y mejorar el proceso de definición de pruebas. Además, permite el entendimiento y la legibilidad de las pruebas ejecutadas por cualquier parte involucrada en el software desarrollado (programadores, analistas, etc.).

Palabras Clave: Lenguaje Específico de Dominio (DSL), Pruebas Unitarias, Metaprogramación en Ruby, Mejorar Proceso Codificación, Pruebas Entendibles y Legibles, Formalización de Pruebas Ejecutadas, DSL interno, Lenguaje Natural.

CONTENIDO

IMÁGENES

TABLAS

1. INTRODUCCIÓN

Una de las claves más importantes a la hora de producir software de calidad es realizar pruebas bajo una amplia variedad de condiciones. Las pruebas no garantizan que el código desarrollado esté libre de errores, pero es un mecanismo para reducir el número de errores. Es importante destacar, que las pruebas permiten la detección de problemas en las fases tempranas del ciclo de desarrollo.

La primera etapa de pruebas a considerar son las pruebas unitarias o pruebas de caja blanca, que nos permiten determinar si un módulo del programa está listo y correctamente terminado. La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto. Una prueba unitaria de calidad debería cumplir las siguientes condiciones:

- Automatizable. No debería requerirse una intervención manual. Esto es especialmente útil para integración continúa.
- Completa. Deben cubrir la mayor cantidad de código.
- Repetible. No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. También es útil para integración continua.
- Independiente. La ejecución de una prueba no debe afectar a la ejecución de otra.
- Reusable. Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Los beneficios que se obtienen con las pruebas unitarias son múltiples destacando los siguientes:

- Simplificación en la Integración: Las pruebas unitarias eliminan las posibles incertidumbres y errores en lo que se espera de cada una de las unidades ayudando a entender la integración de cada una de las partes.
- Refactorización o Regresión: Una vez refactorizado el código; las mismas pruebas unitarias nos pueden servir para probar el nuevo código asegurándonos de que este sigue siendo válido bajo la nueva implementación.
- Documentación: Las pruebas unitarias sirven como método de documentación mismo. Los desarrolladores pueden ver a través de las pruebas unitarias cual es el objetivo de las distintas partes del código de una manera básica.
- Diseño: Cuando se desarrolla el software las pruebas unitarias pueden tomar el lugar del diseño formal. Cada prueba unitaria puede ser vista como un elemento de diseño que especifica las clases, los métodos y el comportamiento observable de la aplicación.

Es importante destacar, las limitaciones de las pruebas unitarias: No descubrirán errores en la integración y en problemas que tengan que ver con el funcionamiento en conjunto del sistema.

Existen multitud de herramientas (también llamados marcos de trabajo) para la realización de pruebas unitarias en los diferentes lenguajes de programación. La definición de las pruebas consiste en la codificación del código fuente necesario para la ejecución automática de dichas pruebas unitarias, utilizando editores de texto plano o editores de código fuente para ello.

La mayoría de los marcos de trabajos son potentes herramientas para la realización de prueba unitarias sobre un código ya desarrollado, dirigidas principalmente a los

desarrolladores. Otros marcos de trabajo, además son herramientas orientadas al desarrollo de software dirigido por comportamiento (BDD, Behavior Driven-Development), dirigidos a los analistas de negocios y los desarrolladores que deben colaborar en el desarrollo del producto, e incluso algunos de ellos, también orientados a utilizar las pruebas unitarias como pruebas de validación del producto.

BDD es una práctica de ingeniería de software basada en TDD (Test Driven Development), la cual es otra práctica de ingeniería del software que involucra otras dos prácticas: *Escribir las pruebas unitarias primero (Test First Development)* y Refactorización (Refactoring). En primer lugar, se escribe una prueba y se verifica que las pruebas fallan. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito.

BDD se centra en el comportamiento de los objetos, es decir, en lo que hacen los objetos en lugar de como se implementen como se hace con TDD. La idea de BDD es que los requisitos sean traducidos a pruebas, es decir, escribir pruebas como especificaciones del comportamiento del sistema. De este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido. Para ello, introduce la idea de usar el lenguaje natural para expresar los escenarios de prueba. En la práctica, se consigue mediante un DSL (Domain-Specific Language o lenguaje específico de dominio), que permite a los expertos del dominio la posibilidad de expresar los escenarios directamente en escenarios de pruebas unitarias.

Un dominio específico tiene un vocabulario especializado para describir cosas que son particulares de ese dominio, con un limitado vocabulario y gramática que es legible, entendible y escribible por expertos del dominio.

Los DSL son comúnmente clasificados como internos o externos:

- Los DSL internos son escritos en un lenguaje de programación de propósito general, cuya sintaxis se ha inclinado a parecerse más al lenguaje natural. Muchos DSL internos envuelven API existentes, librerías o código de negocio para proveer un contenedor con un acceso más alucinante a sus funcionalidades. Son ejecutados con sólo correrlos. Dependiendo en la implementación y el dominio, son usados para construir estructuras de datos, definir dependencias, ejecutar procesos o tareas, comunicarse con otros sistemas o validar entradas de usuario. La sintaxis de un DSL interno está contenida en el lenguaje anfitrión.
- Los DSL externos son expresiones gráficas o textuales de un lenguaje, aunque los DSL textuales tienden a ser más comunes que los gráficos. Las expresiones textuales pueden ser procesadas por una cadena de herramientas que incluyen léxico, un analizador, un transformador de modelo, generadores, y cualquier otro tipo de pos-procesamiento. Los DSL externos son frecuentemente leídos en modelos internos, los cuales forman los fundamentos para su posterior procesamiento. Es útil definir una gramática (por ejemplo, en EBNF). Una gramática provee un punto de partida para la generación de partes de la cadena de herramientas (por ejemplo, editor, visualizador, generador de analizadores). Para los DSL sencillos, un analizador hecho a mano podría ser suficiente; usando, por ejemplo, expresiones regulares. Los analizadores personalizados pueden llegar a ser difíciles de manejar si se espera mucho de ellos, así que tiene sentido mirar las herramientas diseñadas específicamente para trabajar con

gramáticas del lenguaje. Es también común el definir DSL externos como los dialectos XML, aunque la legibilidad es frecuentemente un problema; sobre todo para los lectores no técnicos.

La creación de DSL internos es más fácil en los lenguajes de programación que ofrecen azúcar sintáctica (construcciones que no aportan nueva funcionalidad a un lenguaje, pero que permiten que sea más fácilmente utilizable por los seres humanos) y posibilidades de formato (ej. Scala) que para otros que no lo hacen (ej. Java). El lenguaje Ruby ofrece azúcar sintáctica y posibilidades de formato, además de ser un lenguaje enfocado a la simplicidad y a la productividad, con una sintaxis elegante y natural, que le hace muy fácil de entender.

El propósito de este TFM es proporcionar un marco de trabajo para la realización de pruebas unitarias en Ruby cuyos objetivos son los siguientes:

- Proporcionar un lenguaje de dominio interno a Ruby para la realización de pruebas unitarias que sea simple y cercano al lenguaje natural, que facilite la realización de pruebas unitarias a programadores y desarrolladores de pruebas unitarias. La idea es usar un lenguaje que permita definir escenarios de prueba fácilmente a programadores y desarrolladores de pruebas unitarias.
- Proporcionar un lenguaje de dominio interno que permita identificar claramente el objetivo de las pruebas a realizar, de manera que las pruebas definidas sean fácilmente entendibles por los programadores, y los desarrolladores de pruebas unitarias. La idea es usar una representación simple que permita definir escenarios de prueba que puedan legibles y entendibles con facilidad a programadores y desarrolladores de pruebas unitarias.
- Proporcionar un mecanismo que permita definir pruebas unitarias que sean entendibles y legibles por todas las personas (tanto personal técnico como no técnicos) involucradas en el desarrollo de un producto software, con el objetivo de mejorar la detección de errores de entendimiento en el producto a desarrollar durante las fases tempranas de desarrollo. Para ello, se requiere que las pruebas realizadas sean entendibles por todas las personas involucradas en el desarrollo del producto software.
- Proporcionar un mecanismo de documentación formal de pruebas unitarias que pueda ser usado para la aceptación del software desarrollado.

2. EL PROBLEMA

2.1 Descripción del Problema

Los marcos de trabajo para la realización de pruebas unitarias en Ruby están dirigidos principalmente a los desarrolladores de código. El proceso de definición de una prueba unitaria consiste en programar el código fuente de la prueba a realizar, utilizando la sintaxis del lenguaje de programación Ruby y las APIs proporcionadas por los marcos de trabajo. Dependiendo del uso al que va destinado el marco de trabajo, el marco de trabajo proporciona un mecanismo para la definición de las pruebas basado en un lenguaje más o menos alejado del lenguaje natural.

La comprensión y legibilidad de las pruebas unitarias definidas no resulta fácil por el personal no está familiarizado con Ruby, ya que se utiliza básicamente la sintaxis del lenguaje Ruby para la definición de las pruebas.

En el proceso de desarrollo de un producto software es muy importante la detección temprana de errores, tanto errores funcionales como errores de entendimiento del producto requerido. Por ello es importante, que las pruebas unitarias realizadas sean fácilmente comprensibles y legibles por las partes involucradas en el desarrollo software (ej. analistas de negocio, programadores, probadores, personal técnico o no técnico, etc.).

El mecanismo ideal para codificar pruebas unitarias que fuesen entendibles por todos los actores implicados en el desarrollo de un producto software y a la vez facilitara el proceso de definición de las pruebas unitarias tanto a los programadores como a cualquier persona involucrada en la realización de pruebas unitarias, sería utilizar el lenguaje natural para ello. En la actualidad no hay ningún marco de trabajo para pruebas unitarias en Ruby que nos permita codificar pruebas unitarias utilizando el lenguaje natural.

Una aproximación a la idea de usar el lenguaje natural para la codificación de las pruebas unitarias, es utilizar lenguajes específicos de dominio (DSL) internos que proporcionen un lenguaje específico cercano al lenguaje natural para la codificación de pruebas unitarias y herramientas que transformen las pruebas definidas mediante el DSL al lenguaje natural.

2.2 Restricciones Introducidas al Problema Genérico

En este TFM nos centraremos en definir un prototipo que nos permita validar los siguientes aspectos en la realización de pruebas unitarias en Ruby:

- La codificación de pruebas unitarias a través de un lenguaje de dominio específico interno a Ruby cuya sintaxis esté lo más cercano posible al lenguaje natural. El objetivo será validar si el DSL definido facilita la codificación de las pruebas unitarias y la comprensión de las pruebas unitarias tanto a los programadores como a los desarrolladores de pruebas unitarias.
- La utilidad de las pruebas realizadas a través del marco de trabajo como mecanismo para la detección de errores de entendimiento en fases tempranas del desarrollo de un producto, al proporcionar un mecanismo donde las pruebas definidas son legibles y compresibles por toda persona involucrada en el desarrollo de un producto software.
- La utilidad de las pruebas definidas a través del marco de trabajo como mecanismo de documentación formal de pruebas unitarias que pueda ser usado para la aceptación del software desarrollado.

Las restricciones tenidas en cuenta para alcanzar los objetivos que se pretenden validar en este TFM son las siguientes:

- El marco de trabajo es un prototipo que permite mostrar la idea desarrollada en este TFM para la realización de pruebas unitarias. No es una herramienta que suministre utilidades para la realización de cualquier prueba.
- El marco de trabajo estará enfocado a la realización de pruebas unitarias sobre ficheros código en Ruby, y no al comportamiento global del sistema.

2.3 Opciones Existentes para Resolver el Problema y sus Limitaciones

Un marco de pruebas unitarias permite escribir pruebas, chequear resultados y la ejecución de las mismas de manera automática. Existen multitud de marcos de trabajo para la realización de pruebas unitarias en Ruby tanto en el ámbito profesional como en el ámbito de la educación, siendo RSpec, Minitest y Test/Unit los más usados.

Los marcos de trabajo de pruebas unitarias se pueden clasificar en dos grupos: marcos de trabajo basados en asertos y marcos de trabajo basados en expectaciones. Los marcos de trabajo basado en expectaciones, suelen ser también herramientas para el desarrollo de software dirigido por comportamiento y proporcionan un mecanismo para la definición de pruebas unitarias más cercano al lenguaje natural que los marcos de trabajos no orientados al diseño dirigido por comportamiento.

También existen DSLs (ej. Coulda, Shoulda, etc.) de pruebas unitarias de Ruby, basados en marcos de trabajo existente, que facilitan la codificación de las pruebas unitarias y ayudan a comprender el objetivo de la prueba.

Las pruebas unitarias están separadas del código que se quiere probar por una serie de razones. La primera, permite que el código permanezca ordenado respecto al código de pruebas, haciéndolo fácil de mantener. La segunda, permite que las pruebas sean eliminadas en el momento de despliegue, ya las pruebas son para el desarrollador y los usuarios no las necesitan. La tercera, permite definir un conjunto de contexto (precondiciones o estado necesario para la prueba) diferente para las pruebas y ejecutar las pruebas de nuevo bajo el nuevo contexto.

En las siguientes secciones, se analizarán diferentes marcos de trabajo de pruebas unitarias en Ruby y DSLs con los siguientes objetivos conocer el mecanismo que proporcionan para la definición de pruebas unitarias y compararlos con el propuesto en este TFM.

Además se detallará un ejemplo de la programación de pruebas unitarias en cada uno de los diferentes marcos de trabajo para un mismo software desarrollado en Ruby, junto con las limitaciones que tiene cada marco de trabajo respecto al problema planteado en este TFM.

A continuación, se detallan las definiciones necesarias para comprender las descripciones de los marcos de trabajos y los DSLs analizados:

- Un paso de prueba es un ejemplo ejecutable de cómo el código puede ser usado y su comportamiento esperado en un contexto determinado. Este término se usa en aquellos marcos de trabajo orientados a la realización de pruebas unitarias en vez a la construcción de software dirigido por pruebas.
- Un ejemplo de código es un ejemplo ejecutable de cómo el código puede ser usado y su comportamiento esperado en un contexto determinado. Este

término se usa en los marcos de trabajo orientados al desarrollo dirigido por pruebas indistintamente del término paso de prueba.

- Un caso de prueba, es un conjunto de acciones a ser testados junto con un conjunto de pasos de prueba. Una suite de pruebas, es una colección de uno o más casos de prueba que demuestran si una aplicación funciona correctamente o no.
- Una suite de pruebas, es una colección de uno o más casos de prueba que demuestran si una aplicación funciona correctamente o no. Una suite de pruebas, puede probar una clase entera, un subconjunto de una clase o una combinación de clases
- Doble de Test es un término genérico para cualquier caso donde se remplaza a un objeto en producción para propósitos de prueba. Normalmente, son objetos que simulan el comportamiento de objetos complejos cuando es imposible o impracticable usar el objeto real en la prueba bien porque todavía no se ha desarrollado o por cualquier otro motivo. Son útiles cuando se están escribiendo ejemplos de código que colabora con otros objetos, de manera que te puedes centrar en escribir los ejemplos sin depender de la implementación de los objetos con los que se colabora.
- Un aserto es una declaración del resultado esperado, es decir, “Yo aserto (afirmar con certeza) que X es igual a Y”. En los marcos de trabajo basados en asertos, el paso de prueba es el nivel más básico de una prueba, y verifica que ciertos asertos se satisfagan.
- Una expectación es una expresión de como el código se espera que se comporte. En los marcos de trabajo basados en expectaciones, el paso de prueba (o ejemplo de código) es el nivel más básico de una prueba, y verifica que ciertas expectaciones se satisfagan. El concepto expectación es semejante al concepto de aserto usado en los marcos de trabajo basados en asertos.

Es importante resaltar, que el objetivo de las siguientes secciones es describir como es el proceso de definición de pruebas en los distintos marcos de trabajo y no una descripción detallada de las utilidades que proporcionan los distintos marcos de marcos para la ejecución y captura de los resultados de las pruebas.

2.3.1 Test/Unit

En las siguientes secciones se describe el mecanismo que proporciona Test/Unit para la definición de pruebas unitarias junto con un pequeño ejemplo, que ayudará a contextualizar y entender el mecanismo de definición de pruebas.

Además, se detallarán las limitaciones de Test/Unit respecto al problema planteado en la investigación.

2.3.1.1 Descripción

Test/Unit es un marco de trabajo, basado en asertos, que permite realizar pruebas unitarias bajo un contexto controlado en trozos pequeños de códigos y siempre referidos a objetos y no al comportamiento global de un sistema.

La idea general detrás de Test/Unit es escribir métodos de pruebas que hacen ciertos asertos sobre el código desarrollado y bajo un determinado contexto de

prueba. Los métodos de pruebas desarrollados pueden ser ejecutados en cualquier momento del desarrollo.

Los asertos son la base de Test-Unit. Un aserto es una declaración del resultado esperado, es decir, “Yo aserto (afirmar con certeza) que X es igual a Y”. Cuando el aserto es ejecutado, si el resultado obtenido no es el resultado esperado, se propaga un error con la información pertinente tal que el desarrollador puede corregir el software o la prueba si fuese errónea.

La clase Test::Unit::TestCase engloba una amplia colección de métodos llamados asertos y otros métodos llamados de contexto que facilitan la realización de las pruebas unitarias. Las pruebas se agrupan en clases definidas como subclases de la clase Test::Unit::TestCase.

Los métodos de contexto permiten definir un mismo contexto para todas las pruebas. El contexto de una prueba se realiza mediante el método “setup” que se ejecuta antes la ejecución de cada método de prueba y/o con el método “teardown” que se ejecuta después de la ejecución de cada método de prueba.

Los métodos asertos permiten validar el comportamiento requerido en una prueba unitaria (Pasa o Falla). En la siguiente tabla, se definen los asertos más relevantes proporcionados por Test-Unit:

Aserto	Descripción
assert (boolean, [msg])	Pasa si "boolean" es el valor "True".
assert_block ([msg])	Pasa si la ejecución del bloque de código devuelve el valor "True". Ej. assert_block "couldn't do the thing" do do_the_thing end
assert_equal (expected, actual, [msg])	Pasa si el valor "expected" es igual al valor "actual". Ej: assert_equal "MY STRING", "my string".upcase
assert_not_equal (expected, actual, [msg])	Pasa si el valor "expected" es igual al valor "actual". Ej: assert_not_equal "some string", 5
assert_match (pattern, string, [msg])	Pasa si el valor "string" se ajusta al patrón definido por el valor "pattern". Ej. assert_match(/\w+/, "five, 6, seven").
assert_not_match (pattern, string, [message])	Pasas si el valor "string" no se ajusta al patrón definido por el valor "pattern" igual al valor "actual". Ej. assert_not_match(/two/, "one 2 three").
assert_nil (object, [message])	Pasa si "object" está vacío. Ej. assert_nil ([1,2]).uniq!)
assert_not_nil (object, [message])	Pasa si "object" no está vacío. Ej. assert_not_nil ([1,2])
assert_in_delta (expected_float, actual_float, delta, [message])	Pasa si "expected_float" y "actual_float" son igual dentro de un "delta" de tolerancia. Ej. assert_in_delta 0.05, (50000.0/10**6), 0.00001.

assert_instance_of (class, object, [message])	Pasa si "object" es una instancia de la clase "class". Ej. assert_instance_of String, "gato".
assert_kind_of (class, object, [message])	Pasa si "object" es un tipo de "class". Ej. assert_kind_of Object, "gato"
assert_same (expected, actual, [message])	Pasa si "actual" es la misma instancia que "expected". Ej. o = Object.new; assert_same o, o
assert_not_same (expected, actual, [message])	Pasa si "actual" not es la misma instancia que "expected". Ej. assert_same Object.new, Object.new
assert_raise (*Exceptions,...) {block}	Pasa si bloque de código "block" genera una de las excepciones listadas en "Exceptions". "Exceptions" es un campo opcional. Ej. assert_raise Runtime_Error, LoadError {raise "Boom!!!"}
assert_nothing_raised (*Exceptions,...) {block}	El aserto se cumple si bloque de código "block" no genera una de las excepciones listadas en "Exceptions". "Exceptions" es un campo opcional. Ej. assert_not_raise {[1.2].uniq}
assert_throws (expected_symbol, [message]) {block}	Pasa si el bloque de código "block" genera el símbolo definido por "expected_symbol". Ej. assert_throws :done {throw : done}
assert_nothing_thrown ([message]) {block}	Pasa si el bloque de código "block" no genera nada. Ej. assert_nothing_thrown {do [1.2].uniq}
assert_respond_to (object, method, [message])	Pasa si un el objeto "object" puede responder al método "method". Ej. assert_respond_to "bugbear", :slice.
assert_send (send_array, [message])	Pasa si el método send devuelve el valor "True". send_array está compuesto por un receptor, un método y los argumentos del método. Ej. assert_send [[1,2], :include?,4]
assert_operator (object1, operator, object2, [message])	Pasa si al comparar los objetos "object1" y "object2" usando el operador "operator" el resultado obtenido es "True". Ej. assert_operator 5, :>=, 4
flunk (msg = "Flunk")	Flunk siempre falla. Ej. flunk "not done testing yet"
Notas: "msg" es un parámetro opcional. Contiene un texto que se usará como mensaje de fallo. Si no se proporciona un msg, Test/Unit usa un mensaje de fallo por defecto. El aserto "assert_block" es la base para construir el resto de los asertos que proporciona	

Test-Unit. Se recomienda usar el aserto "assert_in_delta" para comparar números en formato punto flotante.

Tabla 1 - Asertos en Test/Unit

Para escribir pruebas con Test/Unit se aconseja seguir los siguentes pasos:

- Asegurarse que Test::Unit está entre las librerías disponibles.
- Requerir “test/unit” en el fichero de pruebas (require ‘test/unit’).
- Crear una clase que sea subclase de Test::Unit::TestCase (las pruebas se agrupan en clases definidas como subclases de Test::Unit::TestCase).
- Empezar el nombre de los métodos de prueba con la palabra “test_”.
- Incluir al menos un aserto en cada método prueba.
- Definir el contexto adecuado (mediante setup y/o teardown) cuando sea necesario.

La estructura y organización de los elementos necesarios para escribir métodos de prueba (ej. test_fail) es la siguiente:

```
require 'test/unit'

class TestStack < Test::Unit::TestCase
        # def setup

        # end

        # def teardown

        # end


        def test_ok
          assert (true, “Assertion was true”)
        end
end
```

Test/Unit permite agrupar casos de prueba y/o pruebas individuales en suite de pruebas, de manera que se puedan ejecutar todas las pruebas de una suite a la vez.

2.3.1.2 Ejemplo

El código a probar corresponde al fichero “Stack.rb” que contiene la clase “Stack” que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
 # Stack - LIFO structure data
   class Stack
        def initialize
                @store = Array.new
        end
        def pop
                @store.pop
```

```
        end
        def push (element)
                @store.push (element)
        end
        def size
                @store.size
        end
    end
```

Las pruebas unitarias que van a realizar se detallan a continuación:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.
- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.
- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

La definición de las pruebas, consiste en la definición (por parte del usuario) del código fuente correspondiente a las pruebas descritas anteriormente. El contenido del fichero con las pruebas definidas es el siguiente:

```
require_relative 'stack.rb'
require 'test/unit'

class TestStack < Test::Unit::TestCase
    def setup
            @stack = Stack.new
    end

    def test_pop_should_return_top_element
            @stack.push (1)
            @stack.push (2)
            @stack.push (3)
            @stack.push (4)
             assert_equal (4, @stack.pop)
    end

    def test_pop_should_remove_top_element
            @stack.push (1)
            @stack.push (3)
            @stack.push (5)
            @stack.push (7)
            @stack.pop
            assert_equal (3, @stack.size)
    end

    def test_push_should_add_one_element
            @stack.push ("a")
            assert_equal (1, @stack.size)
    end

    def test_push_should_add_the_top_element
```

```
        @stack.push ("a")
        @stack.push ("b")
        assert_equal ("b", @stack.pop)
    end

end
```

2.3.1.3 Limitaciones

Test/Unit proporcionan un mecanismo ágil y rápido para la realización de pruebas unitarias, pero con el inconveniente que la forma de escribir las pruebas unitarias se aleja bastante del lenguaje natural.

El proceso de definición de las pruebas consiste en programar código fuente, utilizando básicamente el lenguaje Ruby y los métodos que proporciona la clase "TestCase". Una prueba es un método que invoca a los métodos asertos proporcionados por la clase "TestCase".

Las pruebas definidas con Test/Unit no son un mecanismo legible y comprensible que permite entender el objetivo de la prueba por todas las partes involucradas en el desarrollo del programa (personal técnico, no técnico, ..). Debido a ello, no es un mecanismo que permita detectar fácilmente errores de entendimiento en el desarrollo de un producto por cualquier persona involucrada en el desarrollo del mismo, durante la fase de pruebas unitarias.

Test/Unit no proporcionan un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas.

2.3.2 RSpec

En las siguientes secciones se describe el mecanismo que proporciona RSpec para la definición de pruebas unitarias junto con un pequeño ejemplo, que ayudará a contextualizar y entender el mecanismo de definición de pruebas.

Además, se detallarán las limitaciones de RSpec respecto al problema planteado en la investigación.

2.3.2.1 Descripción

RSpec es un marco de trabajo basado en expectaciones que permite realizar pruebas unitarias bajo un contexto controlado en trozos pequeños de códigos y siempre referidos a objetos y no al comportamiento global de un sistema.

RSpec también se usa como herramienta para el desarrollo de software dirigido por comportamiento (BDD). También soporta los Dobles de Test, que son objetos que fingen o simulan el comportamiento de objetos reales de una forma controlada).

RSpec proporciona un lenguaje específico (DSL) para definir el comportamiento de objetos, basado en expectaciones. Sus creadores afirman que usa un lenguaje que permite describir comportamiento de la manera que podemos expresarlo como si estuviésemos hablando al cliente o a otro desarrollador. La siguiente imagen presenta un ejemplo de ello.

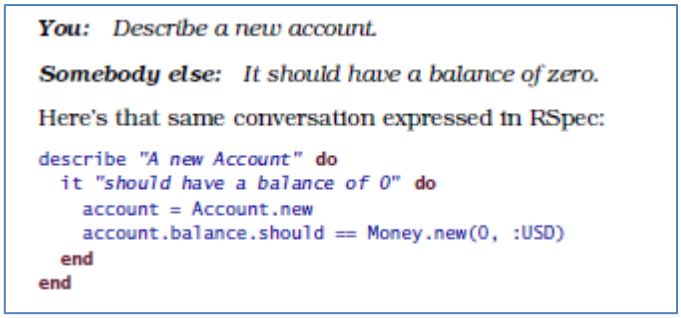


Imagen 1 – Ejemplo RSpec

Dicho DSL está formado por componentes comandos, conectores y expectaciones que permiten la definición del comportamiento esperado del software. Todos ellos están implementados en RSpec como métodos y por lo tanto, se pueden invocar desde cualquier objeto.

Comandos: describe, it y context.

El método "describe(args*, text, &block)" se usa para definir grupos de ejemplos (o casos de pruebas). El método toma un número arbitrario de argumentos y un bloque de código opcional. Los argumentos del método definen:

- El primer argumento puede ser una referencia a una clase o módulo o un texto.
- El segundo argumento es opcional y debe ser un texto, que describe el aspecto del sistema que se quiere probar.
- Un bloque de código, opcional, que define el caso de prueba.

El método "describe" puede ser anidado (describe() dentro de otro describe()) para facilitar la descripción de diferentes casos y estados.

El método "context (args*, text, &block)" es un alias del método "describe", de manera que se pueden usar de manera indistinta.

El método "it (text, &block)" define un ejemplo de código (o paso de prueba), donde un ejemplo es un ejecutable de código de como el código desarrollado debe comportarse. Los argumentos del método definen:

- Texto que es el nombre del ejemplo de código (o paso de prueba) y describe la prueba que se va a realizar.
- Un bloque de código, opcional, que define el paso de prueba.

El texto usado como argumento en los métodos "describe" e "it" permite nombrar y organizar las pruebas de manera significativa.

Conectores: before, let, after y around.

Los conectores definen bloques de código que se ejecutan antes o después de los ejemplos de código.

El método "before(scope, &block)" permite la ejecución de un bloque de código (definido mediante el parámetro "&block") antes de un conjunto de ejemplos de prueba (o paso de prueba). El parámetro "scope" puede tener los valores ":each", ":all" or ":suite":

- El valor ":each" indica que se ejecutará el bloque de código una vez antes de cada ejemplo de prueba.

- El valor ":all" indica que el bloque de código se ejecutará solo una vez antes de todos los ejemplos de prueba de un caso de prueba.
- El valor ":suite" indica que el bloque de código se ejecutará solo una vez antes de todos los ejemplos de prueba de una suite de prueba.

En general, se recomienda usar "before(:each)" ya que recrea el contexto necesario antes de cada ejemplo de prueba y mantiene el estado entre ejemplos de prueba.

El método "let(scope, &block)" es similar al conector "before", pero se usa cuando el bloque consiste sólo en crear variables e inicializarlas.

El método "after(scope, &block)" permite la ejecución de un bloque de código (definido mediante el parámetro "&block") después de un conjunto de ejemplos de prueba. El parámetro "scope" puede tener los valores ":each", ":all" or ":suite":

- El valor ":each" indica que se ejecutará el bloque de código una vez después de cada ejemplo de prueba.
- El valor ":all" indica que el bloque de código se ejecutará solo una vez despues de todos los ejemplos de prueba de un caso de prueba.
- El ":suite" indica que el bloque de código se ejecutará solo una vez despues de todos los ejemplos de prueba de una suite de prueba.

El método "around (&block)" permite envolver comportamiento (definido mediante el parámetro "&block") alrededor de ejemplos de prueba. Es especialmente útil al trabajar con librerías que exponen facilidades a través de métodos que aceptan un bloque como argumento. El uso más común son las transacciones de base de datos.

Expectaciones:

Una expectación es una declaración que expresa en un punto específico de la ejecución de un ejemplo de código, el estado en debería estar un objeto.

Cuando una expectación se ejecuta y el resultado obtenido no es el resultado esperado, se propaga un error con la información pertinente tal que el desarrollador puede corregir el software o la prueba si fuese errónea.

Una buena práctica del uso de las expectaciones, es limitar el número de expectaciones por ejemplo de código, preferiblemente a una. La idea es que un ejemplo de código sea fácil de escribir y mantener limitando el alcance de cada ejemplo.

RSpec logra un alto nivel de expresividad y legibilidad a través un amplio conjunto de expectaciones que proporciona. Para ello, define los métodos "should", "should_not", que reciben bloques de código como argumento, y los combina con los métodos proporcionados por las librerías estándares de Ruby (e.j. mach, equal). y con otros proporcionados por RSpec.

En la siguiente tabla, se resumen las expectaciones proporcionadas por Rspec basadas en la sintaxis "should":

Expectación	Descripción
actual.should equal (expected)	Pasa si: actual.equal?(expected)
actual.should eql (expected)	Pasa si: actual.eql?(expected)
actual.should ==expected	Pasa si: actual == expected"

actual.should_not equal (expected)	Pasa excepto si: actual.equal?(expected)
actual.should_not eql (expected)	Pasa excepto si: actual.eql?(expected)
actual.should_not ==expected	Pasa excepto si: actual ==(expected)
actual.should be_(expected)	Pasa si: actual.expected?
actual.should be_a_(expected)	Pasa si: actual.expected?
actual.should be_an_(expected)	Pasa si: actual.expected?
actual.should_not be_(expected)	Pasa excepto si: actual.expected?
actual.should_not be_a_(expected)	Pasa excepto si: actual.expected?
actual.should_not be_an_(expected)	Pasa excepto si actual.expected?
actual.should match (expected)	Pasa si: actual.match?(expected)
actual.should_not match_(expected)	Pasa excepto si: actual.match?(expected)
actual.should be < expected	Pasa si: actual < expected
actual.should be< = expected	Pasa si: actual <=expected
actual.should be >= expected	Pasa si: actual >= expected
actual.should be > expected	Pasa si: actual > expected
actual.should include (expected)	Pasa si: actual.include? (expected)
actual.should have (n).items	Pasa si: actual.items.length== n
actual.should have_exactly(n).items	Pasa si: actual.items.length== n
actual.should have_at_least(n).items	Pasa si: actual.items.length >= n
actual.should have_at_most(n).items	Pasa si: actual.items.length <= n
actual.should_not include (expected)	Pasa excepto si: actual.include? (expected)
actual.should_not have(n).items	Pasa excepto si: actual.items.length== n
proc.should raise_error	Pasa si genera un error.
proc.should raise_error (message)	Pasa si se genera un error con un mensaje específico.
proc.should raise_error (type)	Pasa si se genera un error de un tipo específico.
proc.should_raise_error (type, message)	Pasa si se genera un error de un tipo y un mensaje específico.
proc.should throw_symbol	Pasa si proc. genera un símbolo.

proc.should throw_symbol (type)	Pasa si proc. genera un símbolo específico..
proc.should_not throw_symbol	Pasa excepto si proc. genera un símbolo.
proc.should_not throw_symbol (type)	Pasa exacto si proc. genera un símbolo específico..
actual.should be_close(expected, delta)	Pasa si actual > (expected - delta) y < (expected + delta)
actual.should_not be_close(expected, delta)	Pasa excepto si actual > (expected - delta) y < (expected + delta)
actual.should respond_to (messages)	Pasa si: messages.each {|m| actual.respond_to?(m)}
actual.should.satisfy {|actual|block}	Pasa si el bloque devuelve verdadero.
actual.should_not satisfy {|actual|block}	Pasa excepto si el bloque devuelve verdadero.
Notas: actual: estado real del objeto. expected: estado esperado del objeto.	

Tabla 2 - RSpec: Expectaciones con sintaxis "Should"

Rspec proporciona los métodos "should" y "should_not" para definir expectaciones sobre cualquier tipo de objeto desde los inicios de Rspec. En las últimas versiones de RSpec, proporciona el método "expect" para definir expectaciones sobre objetos. RSpec recomienda usar "expect" en lugar "should" y "should_not" ya que proporciona mejor legibilidad en los bloques de expectaciones.

Al igual que los métodos "should", "should_not", el método "expect" recibe un bloque de código como argumento, y se combina con los métodos proporcionados por las librerías estándares de Ruby (e.j. mach, equal) y con otros proporcionados por RSpec.

En la siguiente tabla, se resumen las expectaciones proporcionadas por Rspec basadas en la sintaxis "expect":

Expectación	Descripción
expect(actual).to eq(expected)	Pasa si: actual == expected
expect(actual).to eql(expected)	Pasa si: actual.eql?(expected)
expect(actual).not_to eql(not_expected)	Pasa si: !(actual.eql?(expected))
expect (actual).to be (expected)	Pasa si: actual.equal?(expected)
expect (actual).to equal (expected)	Pasa si: actual.equal?(expected)
expect (actual).to be > expected	Pasa si: actual > expected
expect(actual).to be >= expected	Pasa si: actual >= expected

expect(actual).to be <= expected	Pasa si: actual < =expected
expect(actual).to be < expected	Pasa si: actual < expected
expect(actual).to match (/expression/)	Pasa si: actual.match?(expression)
expect(actual).to be_an_instance_of (expected)	Pasa si: actual.class == expected
expect(actual).to be_a(expected)	Pasa si: actual.kind_of?(expected)
expect(actual).to be_an(expected)	Pasa si: actual.kind_of?(expected)
expect(actual).to be_a_kind_of(expected	Pasa si: actual.kind_of?(expected)
expect(actual).to be_truthy	Pasa si: actual == true |nil.
expect(actual).to be true	Pasa si: actual == true
expect(actual).to be_falsy	Pasa si: actual == false |nil.
expect(actual).to be false	Pasa si: actual == true |nil.
expect(actual).to be_nil	Pasa si "actual" es nulo.
expect(actual).to_not be_nil	Pasa si "actual" no es nulo.
expect { ... }.to raise_error	Pasa si genera un error.
expect { ... }.to raise_error(ErrorClass)	Pasa si se genera un error de un tipo específico.
expect { ... }.to raise_error("message")	Pasa si se genera un error con un mensaje específico.
expect { ... }.to raise_error(ErrorClass, "message")	Pasa si se genera un error de un tipo y un mensaje específico.
expect(actual).to be_xxx	Pasa si: actual.xxx?
expect(actual).to have_xxx(:arg)	Pasa si: actual.has_xxx?(:arg)
expect(actual).to include(expected)	Pasa si: actual.include? (expected)
expect(actual).to start_with(expected)	Pasa si: actual.start? (expected)
expect(actual).to end_with(expected)	Pasa si: actual.end? (expected)
Notas: actual: estado real del objeto. expected: estado esperado del objeto. Ejemplos: expect([1, 2, 3]).to include(1) expect([1, 2, 3]).to include(1, 2) expect([1, 2, 3]).to start_with(1) expect([1, 2, 3]).to start_with(1, 2) expect([1, 2, 3]).to end_with(3) expect([1, 2, 3]).to end_with(2, 3)	

```
expect({:a => 'b'}).to include(:a => 'b')
expect("this string").to include("is str")
expect("this string").to start_with("this")
expect("this string").to end_with("ring")
```

Tabla 3 - Rspec: Expectaciones con sintaxis "expect"

2.3.2.2 Ejemplo

El código a probar corresponde a una clase “Stack” (almacenado en el fichero stack.rb) que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
 # Stack - LIFO structure data
   class Stack
       def initialize
              @store = Array.new
       end
       def pop
              @store.pop
       end
       def push (element)
              @store.push (element)
       end
       def size
               @store.size
       end
 end
```

A continuación, se detallan las pruebas unitarias a realizar sobre la clase Stack:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.
- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.
- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

La definición de las pruebas unitarias consiste en definir desde un editor de texto plano o editor de código el siguiente código fuente:

```
require "rspec"
require "stack.rb"

describe Stack do
        before (:each) do
```

```
            @stack = Stack.new
        end
        describe "#pop" do
            before (:each) do
                @stack = ["a", "b", "c", "d"]
            end
            it "should return the top element" do
                @stack.pop.should equal ("d")
            end
            it "should remove the top element" do
                @stack.pop
                @stack.size.should equal (3)
            end
        end
        describe "#push" do
            it "should add the top element" do
                @stack.push ("a")
                @stack.size.should equal (1)
            end
            it "should add the top element" do
                @stack.push ("a")
                @stack.push ("b")
                @stack.pop.should equal ("b")
            end
        end
    end
```

2.3.2.3 Limitaciones

RSpec proporciona una mayor legibilidad y compresión de las pruebas unitarias realizadas respecto a los marcos de trabajo basados en asertos. No obstante, presenta el inconveniente que todavía está alejado del lenguaje natural.

Las pruebas definidas con RSpec no son un mecanismo legible y comprensible que permite entender el objetivo de la prueba por todas las partes involucradas en el desarrollo del programa (personal técnico, no técnico, ..). Debido a ello, no es mecanismo que permita detectar errores de entendimiento en el desarrollo de un producto por cualquier persona involucrada en el desarrollo del mismo durante la fase de pruebas unitarias.

RSpec no proporcionan un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas.

2.3.3 Minitest

En las siguientes secciones se describe el mecanismo que proporciona Minitest para la definición de pruebas unitarias junto con un pequeño ejemplo, que ayudará a contextualizar y entender el mecanismo de definición de pruebas.

Además, se detallarán las limitaciones de Minitest respecto al problema planteado en la investigación.

2.3.3.1 Descripción

Minitest es un marco de trabajo que permite realizar pruebas unitarias basadas en asertos (Minitest/Unit) o basadas en expectaciones (Minitest/Spec), bajo un contexto controlado en trozos pequeños de códigos y siempre referidos a objetos y no al comportamiento global de un sistema.

Minitest soporta el desarrollo de software por comportamiento (BDD), además de la realización de pruebas unitarias sobre código ya implementado. También soporta Dobles de Test (objetos que fingen o simulan el comportamiento de objetos reales de una forma controlada) y Benchmarking (pruebas para valorar las performances de los algoritmos de manera repetible).

2.3.3.1.1 Minitest/Unit

La realización de pruebas unitarias con Minitest/Unit sigue los mismos principios que el marco de trabajo Test-Unit. Al igual que Test-Unit, la idea general detrás de Minitest/Unit es escribir pruebas, donde cada paso de prueba corresponde a un método que hace ciertos asertos sobre el código desarrollado bajo un contexto determinado. Las pruebas desarrolladas son métodos que pueden ser ejecutados en cualquier momento del desarrollo.

La realización de pruebas unitarias con Minitest/unit es muy similar a realizarlas con Test-Unit. Todo lo que se tiene que conocer son los módulos que se requieren, como se nombran las clases (casos de test) y los métodos (pasos de test), y los métodos (aserciones) que se necesitan para la realización de cada test.

La clase Minitest::Unit::TestCase engloba una amplia colección de métodos llamados asertos y otros métodos llamados de contexto que facilitan la realización de las pruebas unitarias. Las pruebas se agrupan en clases definidas como subclases de la clase Minitest::Unit::TestCase.

Minitest proporciona métodos para establecer el contexto del caso de prueba. Para ello proporciona los siguientes métodos:

- setup (&block): El método se ejecuta antes de cada paso de prueba y en él se definen las acciones comunes a ejecutar antes de cada test.
- add_setup_hook (arg =nil, &block). Equivalente al método setup. Permite incluir un bloque de código que se ejecutará antes de cada paso de prueba.
- teardown(&block): Ejecutado después de cada test y en él se definen las acciones comunes a ejecutar después de cada paso de prueba.
- add_teardown_hook (arg =nil, &block). Equivalente al método teardown. Permite incluir un bloque de código que se ejecutará después de cada paso de prueba.

Los métodos asertos permiten validar el comportamiento requerido en una prueba unitaria (Pasa o Falla). En la siguiente tabla, se definen los asertos más relevantes proporcionados por Minitest/Unit. La mayoría de ellos aceptan (opcionalmente) un mensaje (msg) como argumento que es impreso en el caso de que la prueba falle.

Aserto	Descripción
assert (boolean, [msg])	Pasa si "boolean" es el valor "True".
assert_block ([msg])	Pasa si la ejecución del bloque de código devuelve el valor "True".
assert_empty (actual, [msg])	Pasa si "actual" es vacío.
assert_equal (expected, actual, [msg])	Pasa si valor "expected" es igual al valor "actual".
assert_in_delta (expected_float, actual_float, delta, [message])	Pasa si "expected_float" y "actual_float" son igual dentro de un "delta" de tolerancia.
assert_in_epsilon (actual, expected, epsilon, [msg])	Pasa si "expected_float" y "actual_float" tienen un error relativo menor que "epsilon".
assert_includes (collection, obj, [msg])	Pasa si "collection" contiene "obj".
assert_instance_of (class, object, [message])	Pasa si "object" es una instancia de la clase "class".
assert_kind_of (class, object, [message])	Pasa si "object" es un tipo de "class".
assert_match (pattern, string, [msg])	Pasa si el valor "string" se ajusta al patrón definido por el valor "pattern" igual al valor "actual".
assert_nil (object, [message])	Pasa si "object" está vacío.
assert_operator (object1, operator, object2, [message])	Pasa si al comparar los objetos "object1" y "object2" usando el operador "operator" el resultado obtenido es "True".
assert_output (stdout = nil, stderr = nil)	Pasa si "stdout" o "stderr" no devuelven el resultado esperado.
assert_raise (*Exceptions,...) {block}	Pasa si bloque de código "block" genera una de las excepciones listadas en "Exceptions". "Exceptions" es un campo opcional.
assert_respond_to (object, method, [message])	Pasa si un el objeto "object" puede responder al método "method".
assert_same (expected, actual, [message])	Pasa si "actual" es la misma instancia que "expected".
assert_send (send_array, [message])	Pasa si cumple si el método send devuelve el valor "True". send_array está compuesto por un receptor, un método y los argumentos del método.
assert_throws (expected_symbol, [message]) {block}	Pasa si el bloque de código "block" genera el símbolo definido por "expected_symbol".
refute (boolean, [msg])	Falla si "Boolean" es "True".
refute_empty (obj, msg = nil)	Falla si "obj" es "vacío".
refute_equal (exp, act, msg =nil)	Falla si "exp" es igual a "act".
refute_in_delta (expected_float, actual_float, delta, [message]))	Falla si "expected_float" es un valor dentro de un "delta" de tolerancia respecto a "actual_float".

refute_in_epsilon (expected, actual, epsilon, [msg])	Falla si "expected" y "actual" tienen un error relativo dentro de epsilon.
refute_includes (collection, obj, [msg])	Falla si "collection" incluye "obj".
refute_instance_of (class, obj, [msg])	Falla si "obj" es una instancia de "class".
refute_kind_of (class, obj, [msg])	Falla si "obj" es un tipo de "class".
refuse_match (pattern, string, [msg])	Falla el valor "string" si no se ajusta al patrón definido por el valor "pattern".
refute_nil (obj, [msg])	Falla si "obj" es vacío.
refute _operator (o1, op, o2, msg =nil)	Falla si al comparar los objetos "object1" y "object2" usando el operador "operator" el resultado obtenido no es "True".
refute_respond_to (object, method, [message]).	Falla si un el objeto "object" responder al método "method".
refute_same (expected, actual, [message]))	Falla si "actual" es la misma instancia que "expected".

Tabla 4 - Asertos en MiniTest

Minitest también proporciona asertos para valorar las performances de un algoritmo (benchmaking). Para ello, proporciona los siguientes métodos:

- assert_performance (validation, &work). Ejecuta el algoritmo recibido en el argumento "work" recopilando los tiempos de cada ejecución. El rango y las veces de ejecución son proporcionados a un determinado procedimiento definido en "validación".
- assert_performance_constant (threshold =0.99, &work). Ejecuta el algoritmo recibido en el argumento "work" y valora si los tiempos de ejecución se ajustan a un ratio constante dentro del rango definido por "threshold".
- assert_performance_exponential (threshold =0.99, &work). Ejecuta el algoritmo recibido en el argumento "work" y valora si los tiempos de ejecución se ajustan a una curva exponencial dentro de un rango de error definido por "threshold".
- assert_performance_linear (threshold =0.99, &work). Ejecuta el algoritmo recibido en el argumento "work" y valora si los tiempos de ejecución se ajustan a una linea recta dentro de un rango de error definido por "threshold".

2.3.3.1.2 Minitest/Spec

Minitest/Spec permite la realización de pruebas unitarias basadas en expectaciones y el desarrollo de software dirigido por comportamiento (BDD). Para ello proporciona un DSL basado en expectaciones (similar a Rspec) que permite definir el comportamiento esperado del software ya codificado o pendiente a codificar en el caso de BDD.

Minitest/Spec proporciona un DSL formado por componentes comandos, conectores, asistentes y expectaciones que permiten la definición del comportamiento esperado del software. Todos ellos están implementados en Minitest como métodos y por lo tanto, se pueden invocar desde cualquier objeto.

Comandos: describe, it**.**

El método "describe (desc = "anonymous", "desc=anonymous", &block)" se usa para definir grupos de ejemplos (o casos de pruebas). Los argumentos del método definen:

- “desc” representa el aspecto del sistema que se quiere probar. Normalmente, solo se usa uno o dos argumentos de este tipo.
- “block” es el bloque de código que define el caso de prueba.

El método puede ser anidado (describe() dentro de otro describe()) para facilitar la descripción de diferentes casos y estados.

El método "it (desc = "anonymous, &block)" define un ejemplo de código (o paso de prueba), donde un ejemplo es un ejecutable de código de como el código desarrollado debe comportarse. Los argumentos del método definen:

- “desc” es el nombre del ejemplo de código (o paso de prueba) y describe la prueba que se va a realizar.
- “block” es el bloque de código que define el paso de prueba.

Conectores: before, after.

Los conectores definen acciones que se ejecutan antes o después de cada ejemplo de código. Son útiles para evitar repeticiones en los ejemplos de código.

El método "before (type = nil, &block)" se ejecuta antes de cada ejemplo. Los argumentos del método definen:

- El argumento type no es usado, y se proporciona por claridad.
- block contiene el bloque de código a ejecutar antes de cada ejemplo de código.

El método "after (type = nil, &block)" se ejecuta antes de cada ejemplo. Los argumentos del método definen:

- El argumento type no es usado, y se proporciona por claridad.
- block contiene el bloque de código a ejecutar después de cada ejemplo de código.

Notad que los conectores before y after puede ser usados multiples veces, sin embargo, es una buena práctica usarlos solo una vez en cada caso de prueba (es decir, usarlo solo una vez en cada nivel de "describe").

Minitest proporciona también un conector especial "after_tests" que permite ejecutar un bloque de código después de la ejecución de la suite de prueba. Este método puede ser invocado varias veces, pero es una buena práctica, solo usarlo una vez con el fin de evitar confusión.

Asistentes: let, subject, specify, skip.

Minitest proporciona una serie de métodos asistentes que facilitar la lectura y escritura de las pruebas.

El método "let (name, &block)" es una simplificada versión del conector "before" y se usa para inicializar accesores (accesores son métodos que permiten leer y escribir el valor de una variable de un objeto instanciado) predefinidos y los valores que devuelven. Los argumentos del método definen:

- “name” es el nombre del accesor.
- “block” contiene el bloque de código a ejecutar asociado al accesor.

El método "subject (&block)" funciona de manera similar al método "let" pero solo se puede usar para definir un accesor llamado "subject". Se usa para especificar el objeto para el cual su comportamiento se está describiendo. El argumento del método es el bloque de código a ejecutar asociado al accesor.

El método specify (desc = "anonymous", &block) es un alias del método it. Se usa cuando no tiene sentido dar una descripción al ejemplo de código.

El método "skip (string)" proporciona la manera de saltar la ejecución de ciertos ejemplos de código. El método, opcionalmente, toma como argumento un texto que se usa para propocionar una explicación de porqué se ha saltado la ejecución de un ejemplo de código.

Otra manera de hacer "skips" es usar el método "it" un bloque de código asociado. Esto puede ser usado para mantener una lista de pruebas que has planeado realizar. De igual manera que con los ejemplos "skip", la ejecución indicará que hay ejemplos pendientes de completar.

Expectaciones:

Una expectación es una declaración que expresa en un punto específico de la ejecución de un ejemplo de código, que algo debería estar en algún estado.

Una buena práctica del uso de las expectaciones, es limitar el número de expectaciones por ejemplo de código, preferiblemente a una. La idea es que un ejemplo de código sea fácil de escribir y mantener limitando el alcance de cada ejemplo.

Minitest proporciona un amplio conjunto de expectaciones. Una de las grandes diferencias con respecto a RSpec, es que las expectaciones usan "must" en lugar de "should" y "wont" en lugar de "should_not".

En la siguiente tabla se detallan las expectaciones proporcionadas por MiniTest/Spec:

Expectaciones	Ejemplos
must_be	labels.size.must_be :==, 0
must_be_close_to	traits.size.must_be_close_to 1,1
must_be_empty	labels.must_be_empty
must_be_instance_of	hipster.must_be_instance_of Hipster
must_be_kind_of	labels.must_be_kind_of Enumerable
must_be_nil	labels.first.must_be_nil
must_be_same_as	traits.must_be_same_as traits
must_be_silent	proc { "no stdout or stderr" }.must_be_silent
must_be_within_epsilon	traits.size.must_be_within_epsilon 1,1
must_equal	traits.size.must_equal 2
must_include	traits.must_include "skinny jeans"
must_match	traits.first.must_match /silly/
must_output	proc { print "#{traits.size}!" }.must_output "2!"
must_respond_to	traits.must_respond_to :count
must_raise	proc { traits.foo }.must_raise NoMethodError
must_send	traits.must_send [traits, :values_at, 0]

must_throw	proc { throw Exception if traits.any? }.must_throw Exception
wont_be	n.wont_be :<=, 42
wont_be_empty	collection.wont_be_empty
wont_be_instance_of	obj.wont_be_instance_of klass
wont_be_kind_of	obj.wont_be_kind_of mod
wont_be_nil	obj.wont_be_nil
wont_be_same_as	a.wont_be_same_as b
wont_be_within_delta	n.wont_be_close_to m [, delta]
wont_be_within_epsilon	n.wont_be_within_epsilon m [, epsilon]
wont_equal	a.wont_equal b
wont_include	collection.wont_include obj
wont_match	a.wont_match b
wont_respond_to	obj.wont_respond_to msg

Tabla 5 - Expectaciones en MiniTest

Benchmarking

Minitest/Spec también permite la realización de pruebas para valorar las performances de un algoritmo. Para ello, proporciona los siguientes métodos:

- bench_performance_constant (name, threshold =0.99, &work). Crea un benchmark que verifica que la performance es constante.
- bench_performance_exponential (name, threshold =0.99, &work). Crea un benchmark que verifica que la performance es exponencial.
- bench_performance_linear (name, threshold =0.99, &work). Crea un benchmark que verifica que la performance es constante.

2.3.3.2 Ejemplo

El código a probar corresponde al fichero “Stack.rb” que contiene la clase “Stack” que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
 # Stack - LIFO structure data
   class Stack
       def initialize
              @store = Array.new
       end
       def pop
              @store.pop
       end
       def push (element)
              @store.push (element)
       end
       def size
```

```
            @store.size
        end
    end
```

A continuación, se detallan las pruebas unitarias a realizar sobre la clase Stack:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.
- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.
- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

Las pruebas unitarias definidas con Minitest/Unit son las siguientes:

```
require_relative 'stack.rb'
require 'minitest/autorun'

class TestStack < Minitest::Test
    def setup
        @stack = Stack.new
    end

    def test_pop_should_return_top_element
        @stack.push (1)
        @stack.push (2)
        @stack.push (3)
        @stack.push (4)
         assert_equal (4, @stack.pop)
    end

    def test_pop_should_remove_top_element
        @stack.push (1)
        @stack.push (3)
        @stack.push (5)
        @stack.push (7)
        @stack.pop
        assert_equal (3, @stack.size)
    end

    def test_push_should_add_one_element
        @stack.push ("a")
        assert_equal (1, @stack.size)
    end

    def test_push_should_add_the_top_element
        @stack.push ("a")
        @stack.push ("b")
        assert_equal ("b", @stack.pop)
    end

end
```

Las pruebas unitarias definidas con Minitest/Spec son las siguientes:

```
require "minitest/autorun”
```

```
require "stack.rb"

describe Stack do
        before do
                @stack = Stack.new
        end
        describe "#pop" do
            before  do
                    @stack = ["a", "b", "c", "d"]
            end
             it "should return the top element" do
                @stack.pop.must_be == "d"
            end
            it "should remove the top element" do
                @stack.pop
                 @stack.size.must_be ==  3
            end
        end
        describe "#push" do
            it "should add the top element" do
                @stack.push ("a")
                @stack.size.must_be == 1
            end
            it "should add the top element" do
                @stack.push ("a")
                @stack.push ("b")
                @stack.pop.must_be == "b"
            end
    end
end
```

2.3.3.3 Limitaciones

Minitest/Unit proporcionan un mecanismo ágil y rápido para la realización de pruebas unitarias, pero con el inconveniente que la forma de escribir las pruebas unitarias se aleja bastante del lenguaje natural.

El proceso de definición de las pruebas consiste en programar código fuente, utilizando básicamente el lenguaje Ruby y los métodos que proporciona la clase "MinitestCase". Una prueba es un método que invoca a los métodos asertos proporcionados por la clase "MinitestCase".

Las pruebas definidas con Test/Unit no son un mecanismo legible y comprensible que permite entender el objetivo de la prueba por todas las partes involucradas en el desarrollo del programa (personal técnico, no técnico,etc.). Debido a ello, no es mecanismo que permita detectar fácilimente errores de entendimiento en el

desarrollo de un producto por cualquier persona involucrada en el desarrollo del mismo, durante la fase de pruebas unitarias.

Minitest/Spec proporciona una mayor legibilidad y compresión de las pruebas unitarias realizadas respecto a los marcos de trabajo basados en asertos. En mi opinión, el DSL todavía está lejos del lenguaje natural y requiere mejoras para acercase más al lenguaje natural.

Las pruebas definidas con Minitest/Spec no son un mecanismo legible y comprensible que permite entender el objetivo de la prueba por todas las partes involucradas en el desarrollo del programa (personal técnico, no técnico,etc.). Debido a ello, no es mecanismo que permita detectar errores de entendimiento en el desarrollo de un producto por cualquier persona involucrada en el desarrollo del mismo durante la fase de pruebas unitarias.

Minitest no proporciona un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas.

2.3.4 Bacon

En las siguientes secciones se describe el mecanismo que proporciona Bacon para la definición de pruebas unitarias junto con un pequeño ejemplo, que ayudará a contextualizar y entender el mecanismo de definición de pruebas.

Además, se detallarán las limitaciones de Bacon respecto al problema planteado en la investigación.

2.3.4.1 Descripción

Bacon es un pequeño clon de RSpec que proporciona las principales características de RSpec. Es un marco de trabajo, basado en expectaciones, que permite la realización de pruebas unitarias en un contexto controlado, en trozos pequeños de códigos y siempre referidos a objetos y no al comportamiento global de un sistema.

Al igual que RSpec, Bacon también es una herramienta para el desarrollo de software dirigido por comportamiento (BDD).

Bacon proporciona un lenguaje específico (DSL) para definir el comportamiento de objetos, basado en expectaciones. Dicho DSL está formado por componentes comandos, conectores y expectaciones que permiten la definición del comportamiento esperado del software. Todos ellos, están implementados como métodos y por lo tanto, se pueden invocar desde cualquier objeto.

Comandos: describe, it y context.

El método "describe (args, &block)" se usa para definir grupos de ejemplos (o casos de pruebas). Toma un número arbitrario de argumentos y un bloque de código opcional. Los argumentos del método definen:

- El primer argumento puede ser una referencia a una clase o módulo o un texto.
- El segundo argumento es opcional y debe ser un texto, que describe el aspecto del sistema que se quiere probar.
- Un bloque de código, opcional, que define el caso de prueba.

El método "describe (args, &block)" puede ser anidado para facilitar la descripción de diferentes casos y estados.

El método "it (description, &block)" define un ejemplo de código (o paso de prueba), donde un ejemplo es un ejecutable de código de como el código desarrollado debe comportarse. Los argumentos del método definen:

- Texto que es el nombre del ejemplo de código (o paso de prueba) y describe la prueba que se va a realizar.
- Un bloque de código, opcional, que define el paso de prueba.

El texto usado como argumento en los métodos describe() e it() permite nombrar y organizar las pruebas de manera significativa.

Conectores: before y after.

Los conectores definen bloques de código que se ejecutan antes o después de los ejemplos de código. Se usan para definir el contexto de una prueba o de un conjunto de pruebas.

El método before(&block) permite la ejecución de un bloque de código (definido mediante el parámetro "&block") una vez antes de cada ejemplo de prueba.

El método after(&block) permite la ejecución de un bloque de código (definido mediante el parámetro "&block") una vez despues de cada ejemplo de prueba.

Expectaciones:

Una expectación es una declaración que expresa en un punto específico de la ejecución de un ejemplo de código, algo debería estar en algún estado.

Una buena práctica del uso de las expectaciones, es limitar el número de expectaciones por ejemplo de código, preferiblemente a una. La idea es que un ejemplo de código sea fácil de escribir y mantener limitando el alcance de cada ejemplo.

Bacon logra un alto nivel de expresividad y legibilidad a través un amplio conjunto de expectaciones que proporciona. Para ello, proporciona los métodos "should", "not", "be", "equal", "identical_to", "match", "satisfy", "change?", "raise?", "throw?", "false?", "true?" "close_to?" y permite su combinación para definir una amplia gama de expectaciones.

En la siguiente tabla se detallan algunos ejemplos de las expectaciones que se pueden definir en Bacon:

Expectación	Descripción
actual.should.equal (expected)	Pasa si "actual.equal?(expected)"
actual.should.not.equal (expected)	Pasa excepto si "actual.equal?(expected)"
actual.should.not.be(expected)	Pasa excepto si "actual.expected?"
actual.should.be.identical_to (expected)	Pasa si "actual.equal?(expected)"
actual.should.not.be.identical_to (expected)	Pasa excepto si "actual.equal?(expected)"
actual.should.be.same_as (expected)	Pasa si "actual.equal?(expected)"
actual.should.not.be.same_as (expected)	Pasa excepto si "actual.equal?(expected)"

actual.should (expected)	Pasa si "actual.expected?"
actual.should.not (expected)	Pasa excepto "actual.expected?"
actual.should.match (expected)	Pasa si "actual.match?(expected)"
actual.should.not.match (expected)	Pasa excepto si "actual.match?(expected)"
should.raise (arg, exceptions){ }	Pasa si se genera un error de un tipo específico.
should.not.raise (exceptions){ }	Pasa excepto si se genera un error de un tipo específico.
.should.throw (symbol){ }	Pasa si proc. genera un símbolo específico.
should.not.throw (symbol){ }	Pasa excepto si proc. genera un símbolo específico.
actual.should.satisfy {|object|}	Pasa si el bloque devuelve verdadero.
Notas: "actual" es el estado real del objeto. "expected" es el estado esperado del objeto.	

Tabla 6 – Expectaciones en Bacon

2.3.4.2 Ejemplo

El código a probar corresponde al fichero “Stack.rb” que contiene la clase “Stack” que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
 # Stack - LIFO structure data
   class Stack
        def initialize
                @store = Array.new
        end
        def pop
                @store.pop
        end
        def push (element)
                @store.push (element)
        end
        def size
                 @store.size
        end
  end
```

A continuación, se detallan las pruebas unitarias a realizar sobre la clase Stack:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.
- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.
- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

Las pruebas unitarias definidas con Bacon son las siguientes:

```
require "bacon"
require "stack.rb"

describe Stack do
        before do
                @stack = Stack.new
        end
        describe "#pop" do
            before (:each) do
                @stack = ["a", "b", "c", "d"]
            end
             it "should return the top element" do
                @stack.pop.should.equal ("d")
            end
            it "should remove the top element" do
                @stack.pop
                @stack.size.should.equal (3)
            end
        end
        describe "#push" do
            it "should add one element" do
                @stack.push ("a")
                @stack.size.should.equal (1)
            end
            it "should add the top element" do
                @stack.push ("a")
                @stack.push ("b")
                @stack.pop.should.equal ("b")
            end
    end
end
```

2.3.4.3 Limitaciones

Bacon proporciona una mayor legibilidad y compresión de las pruebas unitarias realizadas respecto a los marcos de trabajo basados en asertos y expectaciones al proporcionar una amplia gama de expectaciones mediante la combinación de los métodos que proporciona.

El objetivo de las pruebas no es fácilmente identificable para los no iniciados en el lenguaje de programación Ruby. Debido a ello, no es un mecanismo que permita detectar errores de entendimiento en el desarrollo de un producto por cualquier persona involucrada en el desarrollo del mismo durante la fase de pruebas unitarias.

Bacon no proporciona un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas.

2.3.5 Wrong

En las siguientes secciones se describe el mecanismo que proporciona Wrong para la definición de pruebas unitarias junto con un pequeño ejemplo, que ayudará a contextualizar y entender el mecanismo de definición de pruebas.

Además, se detallarán las limitaciones de Wrong respecto al problema planteado en la investigación.

2.3.5.1 Descripción

Wrong es un marco de trabajo basado en asertos que permite realizar pruebas unitarias en un contexto controlado, en trozos pequeños de códigos y siempre referidos a objetos y no al comportamiento global de un sistema. Wrong, a diferencia de otros marcos de trabajo basado en asertos, proporciona solamente un método aserto general que toma un bloque predicado como argumento y mensajes de fallo de asertos muy ricos en detalles.

La mayoría de los marcos de trabajo en asertos proporcionan muchos asertos y mensajes de error escasos de contenido. La idea general de Wrong, es reemplazar todos los incontables asertos “assert_this”, "assert_that" (los cuales solo existen para dar un mensaje de fallo útil que no sea tan simple como “assertion failed”) por uno más genérico y proporcionar mensajes de error más útil que el típico "assertion failed". Para ello suministra un simple método para asertos positivos (la prueba pasa si resultado del valor "True") y un simple método para asertos negativos (la prueba pasa si resultado del valor "False"), tomando ambos un bloque de código como argumento.

En la siguiente tabla se detallan los asertos proporcionados por Wrong:

Aserto	Descripción
assert (*args, &block)	Pasa si el resultado de la ejecución del bloque de código devuelve el valor "True"
deny (*args, &block)	Falla si el resultado de la ejecución del bloque de código devuelve el valor "True"
block es la prueba a realizar.	

Tabla 7 - Asertos Wrong

Wrong también suministra varios métodos asistentes (rescuing, capturing, close_to) para facilitar la definición de pruebas unitarias:

- El método rescuing permite capturar los errores producidos en la ejecución (e.j. assert {rescuing {raise "vanilla"}.message == "chocolate").
- El método capturing permite capturar los flujos de salida de Ruby (e.j. assert {capturing {puts "hi"} == "hi\n"}).
- El método close_to permite comparar números flotantes (ej. assert {5.0.closed_to?(5.001)}. También se usar junto con los métodos Time, Date y DateTime de Ruby, los cuales proporcionan tiempos y fechas.

Existen adaptadores que permiten el uso de Wrong junto con otros marcos de trabajo. Actualmente, se ha desarrollado adaptatores para los siguientes marcos de trabajo:

- Test-Unit (require 'wrong/adapter/test_unit').
- Minitest (''require 'wrong/adapter/minitest').
- RSpec (require 'wrong/adapter/rspec').

A continuación, un ejemplo del uso del adaptador Wrong para Test-Unit:

```
require "test/unit"
require "wrong/adapters/test_unit"
include "Wrong"
class PlusTest < Test::Unit::TestCase
        def test_adding_two_and_two
                assert { 2 + 2 == 4 }
        end
end
```

A continuación, un ejemplo del uso del adaptador Wrong para RSpec:

```
require "rspec"
require "wrong/adapters/rspec"
include "Wrong"
describe "plus" do
        it "adds two and two" do
                assert { 2 + 2 == 4 }
        end
end
```

2.3.5.2 Ejemplo

El código a probar corresponde al fichero “Stack.rb” que contiene la clase “Stack” que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
 # Stack - LIFO structure data
   class Stack
        def initialize
                @store = Array.new
```

```
    end
    def pop
        @store.pop
    end
    def push (element)
        @store.push (element)
    end
    def size
        @store.size
    end
end
```

A continuación, se detallan las pruebas unitarias a realizar sobre la clase Stack:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.
- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.
- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

Las pruebas unitarias definidas con Wrong sin uso de adaptadores son las siguientes:

```
requiere "wrong"
require "stack.rb"
include "Wrong"
#----test 1
@stack1 = Stack.new
@stack1 = ["a", "b", "c", "d"]
assert {@stack1.pop == "d"}
#----test 2
@stack2 = Stack.new
@stack2 = ["a", "b", "c", "d"]
@stack2.pop
assert {@stack2.size== 3}
#-----test3
@stack3 = Stack.new
@stack3.push ("a")
@stack3.pop
assert {@stack3.size== 1}
#---test4
```

```
@stack4 = Stack.new
@stack4.push ("a")
@stack4.push ("b")
assert {@stack4.pop == "b"}
```

2.3.5.3 Limitaciones

Wrong proporciona un mecanismo ágil y rápido para la realización de pruebas unitarias al reducir el número de asertos y proporcionar mensajes de error más detallados. No obstante, el mecanismo que proporciona la definición de las pruebas unitarias se aleja bastante del lenguaje natural.

El uso de Wrong junto a los adaptadores de "Test-Unit" o "RSpec" tampoco introduce ninguna mejora al uso de Wrong sin dichos adaptadores, respecto a la definición de pruebas con un lenguaje cercarno al lenguaje natural.

Las pruebas definidas con Wrong no son un mecanismo legible y comprensible que permite entender el objetivo de la prueba por todas las partes involucradas en el desarrollo del programa (personal técnico, no técnico, etc.). Debido a ello, no es mecanismo que permita detectar fácilimente errores de entendimiento en el desarrollo de un producto por cualquier persona involucrada en el desarrollo del mismo, durante la fase de pruebas unitarias.

Wrong no proporciona un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas.

2.3.6 Riot

En las siguientes secciones se describe el mecanismo que proporciona Riot para la definición de pruebas unitarias junto con un pequeño ejemplo, que ayudará a contextualizar y entender el mecanismo de definición de pruebas.

Además, se detallarán las limitaciones de Riot respecto al problema planteado en la investigación.

2.3.6.1 Descripción

Riot es un marco de trabajo, basado en asertos, que permite realizar pruebas unitarias bajo un contexto controlado en trozos pequeños de códigos y siempre referidos a objetos y no al comportamiento global de un sistema. El objetivo de Riot se centra en permitir realizar pruebas de manera rápida. También soporta los Dobles de Test, que son objetos que fingen o simulan el comportamiento de objetos reales de una forma controlada.

A diferencia de otros marcos de trabajo, no proporciona métodos para definir el contexto de un grupo de pruebas al estilo de los métodos "setup" y “teardown” de Unit-Test que se ejecutan antes de cada prueba y después de cada prueba de un caso de prueba.

Cada prueba consiste en un bloque "context (descripción, &definición)" formado por uno o varios asertos ("asserts" o "denies") respecto al objeto bajo prueba. Cada prueba se ejecuta en un determinado contexto y el bloque "setup" solo se ejecuta una vez por contexto.

En el siguiente ejemplo se muestra la estructura de una prueba en Riot.

```
context "An empty Array" do
    setup { Array.new }
    asserts ("it is empty") { topic.empty? }
```

```
        denies ("it has any elements") { topic.any? }
    end # An Array
```

En el ejemplo se observa que el bloque "setup" no usa variables de instancia para almacenar el objeto bajo prueba. Para acceder al objeto bajo prueba en los bloques asertos se usa el atributo "topic". Cada aserto contiene un bloque de código que define la prueba a realizar. El resultado de un aserto siempre es "True" o "False".

Cuando se usa el aserto "asserts", el resultado "True" indica que la prueba a pasado y el resultado "False" indica que la prueba a fallado. Cuando se usa el aserto "denies", el resultado "True" indica que la prueba a fallado y el resultado "False" que la prueba ha pasado.

Riot permite el anidamiento de contextos ("context" dentro de otro "context").

Junto a los asertos "asserts" y "denies", Riot permite crear asertos más complejos mediante la construcción de macros de asertos. Hay dos formas de pasar argumentos a una macro: una a través de argumentos de entrada o derivado del suministrado de la ejecución de bloque de código.

Riot proporciona las siguientes utilidades para la construcción de macros de asertos junto con los asertos "asserts" y "denies":

Utilidad	Descripción
equals	Compara la igualdad entre el valor real y el valor esperado usando el operador "==" Ej. assert.equals{object}
equivalent_to	Compara la equivalencia entre el valor real y el valor esperado usando el operador "===". Ej.assert.equivalent_to{object}
assigns	Chequea que el valor real tiene una variable instancia definidad dentro su alcance
nil	Chequa si el valor real es nulo. Ej.assert.nil
matches	Compara el valor real a una expresión regular.
raises	Valida que el tipo de excepción generada por un bloque de código. Opcionalmente, se puede incluir como argumento el mensaje a presentar. Ej. assert.raises (ExceptionClass, "Expected message".
responds_to	Chequea que el valor real "respond_to?" a un mensaje particular. Ej.denies.respond_to ("foo").
includes	Chequea la existencia de un carácter o secuencia de caracteres en un "string",

	un elemento en un "array" y una "clave" en un índice. Ej.assert("this array"){[1,2,3]}.includes(2)
size	Compara el tamaño de objeto real a un número que se suministra como argumento. Funciona con cualquier cosa que responda con un valor numérico ("string", "arrays",etc.).
empty	Chequea que el resultado obtenido de la invocacion a "empty?" sobre un objeto. Ej.assert.empty.
same_elements	Compara el objeto real y el esperado para ver si contienen los mismos elementos. Ej.assert_same_elements {array}.

Tabla 8 - Utilidades Riot

Riot también soporta los asertos "should" de manera equivalente a "asserts" y "should_not" de manera equivalente a "denies".

2.3.6.2 Ejemplo

El código a probar corresponde al fichero “Stack.rb” que contiene la clase “Stack” que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
# Stack - LIFO structure data
  class Stack
      def initialize
            @store = Array.new
      end
      def pop
            @store.pop
      end
      def push (element)
            @store.push (element)
      end
      def size
             @store.size
      end
end
```

A continuación, se detallan las pruebas unitarias a realizar sobre la clase Stack:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.

- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.
- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

Las pruebas unitarias definidas con Riot son las siguientes:

```
requiere "riot"
require "stack.rb"
context "tests 1,2,3,4" do
        setup {Stack.new}
        setup {topic = ["a", "b", "c", "d"]}
        asserts ("test 1:it should return the top element") {topic.pop == "d"}
        asserts ("test 2: it should remove the top element") {topic.size(3)}
        asserts ("test 3: it should add one element") do
                topic.push ("e")
                topic.pop == "e"
        end
        asserts ("test 4: it should add the top element") do
                topic.push ("e")
                topic.push ("f")
                topic.pop == "f"
        end
end
```

2.3.6.3 Limitaciones

Riot proporcionan un mecanismo ágil y rápido para la realización de pruebas unitarias, pero con el inconveniente que la forma de escribir las pruebas unitarias se aleja bastante del lenguaje natural.

Las pruebas definidas con Riot no son un mecanismo legible y comprensible que permite entender el objetivo de la prueba por todas las partes involucradas en el desarrollo del programa (personal técnico, no técnico, etc.). Debido a ello, no es mecanismo que permita detectar fácilmente errores de entendimiento en el desarrollo de un producto por cualquier persona involucrada en el desarrollo del mismo, durante la fase de pruebas unitarias.

Riot no proporciona un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas.

2.3.7 Cucumber

En las siguientes secciones se describe el mecanismo que proporciona Cucumber para la realización de pruebas unitarias, junto con un ejemplo de pruebas de unitarias para una pequeño trozo de código que permitirá entender el mecanismo de pruebas unitarias proporcionado por Cucumber.

Además, se detallarán las limitaciones de Cucumber respecto al problema planteado en la investigación.

2.3.7.1 Descripción

Cucumber es un marco de trabajo para el desarrollo de software dirigido por comportamiento (BDD). También es una herramienta para la realización de pruebas unitarias sobre trozos de código y sobre aplicaciones enteras.

Usa un lenguaje simple para describir escenarios que pueden ser escritos y entendidos por personal técnico y no técnico. Los escenarios que se definen con Cucumber representan las pruebas de aceptación del cliente y son usados para automatizar el sistema que están siendo desarrollando. Cucumber permite describir las características de las aplicaciones en un formato de texto plano y usarlos para interactuar automáticamente con la aplicación.

Cucumber proporciona un formato estándar para expresar requisitos en forma de características y escenarios que se puede usar para automatizar las pruebas del sistema que se está probando. El lenguaje Gherkin proporciona una estructura común y una variedad de herramientas para describir características.

Una característica (feature) está compuesta por un título (identifica en pocas palabras una actividad para la cual un usuario emplea al sistema), una breve narración (narración para soportar el contexto para los escenarios ejecutables que se tienen que escribir) y por un número arbitrario de escenarios. Los escenarios están compuestos de pasos que empiezan con “Given”, “When”, “Then”, “And” or “But”.

Como ya se ha mencionado, las características se escriben en un lenguaje simple llamado Gherkin (DSL externo), que es interpretado por Cucumber. Dicho lenguaje está formado por un conjunto reducido de palabras clave:

- Feature
- Background
- Scenario
- Scenario outline
- Scenarios
- Given
- When
- Then
- And (or but)
- | (usado para definir tablas)
- “” (usado para definir texto en múltiples líneas)
- # (usado para comandos).

Tras estas palabras claves se puede escribir cualquier cosa.

Cada fichero de características (features) debe empezar con la palabra clave seguido por “:” y una descripción. La descripción puede ser de varias líneas, y después una breve narración.

```
# language: en
Feature: Compute distance
  In order to calculate fuel consumption
  As a driver
  I want to see the total distance of a trip
```

Imagen 2 – Ejemplo Característica (Feature)

Es importante resaltar que el texto escrito en la narración no es validado por Cucumber. La razón para escribir esta parte es para facilitar la compresión del propósito de la prueba.

Los escenarios son concretos ejemplos de cómo el sistema se debe comportar. Los escenarios permiten contestar las cuestiones relacionadas a que debería ocurrir bajo ciertas circunstancias. La primera parte del escenario es la palabra clave "Scenario", seguido por ":" y un nombre que describe el escenario en una sentencia. Cada escenario está compuesto por un número arbitrario de pasos que describen cualquier cosa que ocurra en el escenario. Un paso, generalmente es una línea de texto que empieza por una de las siguientes palabras claves: Given, When, Then, And, and But.

```
Scenario: transfer money (declarative)
  Given I have $100 in checking
  And I have $20 in savings
  When I transfer $15 from checking to savings
  Then I should have $85 in checking
  And I should have $35 in savings

Scenario: transfer money (imperative)
  Given I have $100 in checking
  And I have $20 in savings
  When I go to the transfer form
  And I select "Checking" from "Source Account"
  And I select "Savings" from "Target Account"
  And I fill in "Amount" with "15"
  And I press "Execute Transfer"
  Then I should see that I have $85 in checking
  And I should see that I have $35 in savings
```

Imagen 3 – Ejemplo Escenarios (Scenario)

Los pasos "Given" se usa para crear el contexto, los pasos "When" para describir un evento que ocurre dentro de ese contexto, y los pasos "Then" para describir los resultados esperados. También, se puede usar "And" y "But", cada uno de los cuales asume la naturaleza de un paso previo. Un paso "And" precedido por un paso "When" se considera otro paso "When". Las definiciones de Paso son equivalentes a declaración de funciones en un lenguaje de programación convencional y se definen en un fichero aparte (fichero de definición de pasos).

Es importante resaltar que es el usuario de Cucumber tiene que definir el fichero de características con los escenarios y pasos deseados, así como las definiciones de pasos correspondientes a los escenarios y los pasos definidos en el fichero de características.

Al ejecutar las pruebas diseñadas (el usuario dispone de una serie de comandos para interactuar con Cucumber), Cucumber valida los pasos en cada escenario e intenta mapearlos a una de las definiciones de paso que hemos escrito en Ruby. Si en encuentra uno, lo ejecuta. Cucumber siempre intenta ser útil y sugiere como se puede implementar una definición de paso siempre que encuentra un paso no definido todavía.

En las siguientes imágenes presentan un ejemplo de cómo Cucumber sugiere una posible implementación de un escenario. Imagen 4 muestra un ejemplo de escenario e Imagen 5 muestra la sugerencia de Cucumber para implementar el Sceanario.

```
# language:en
Feature: Traveler books room
  In order to reduce staff
  As a hotel owner
  I want travelers to book rooms on the web

  Scenario: Successful booking
    Given a hotel with "5" rooms and "0" bookings
```

Imagen 4 – Scenario

```
# language:en
Feature: Traveler books room
  In order to reduce staff
  As a hotel owner
  I want travelers to book rooms on the web

  Scenario: Successful booking                    # features/book_room.feature:7
    Given a hotel with "5" rooms and "0" bookings # features/book_room.feature:8

1 scenario (1 undefined)
1 step (1 undefined)
0m0.001s

You can implement step definitions for undefined steps with these snippets:

Given /^a hotel with "([^"]*)" rooms and "([^"]*)" bookings$/ do |arg1, arg2|
  pending # express the regexp above with the code you wish you had
end
```

Imagen 5 – Sugerencia de Implementación

Cucumber también dispone de conectores (solo visibles al personal técnico que elabora las pruebas), que permite la ejecución de acciones comunes antes y después de cada escenario, y después de cada paso de un escenario:

- Before: Ejecutado antes de cada escenario.
- After: Ejecutado después de cada escenario.
- AfterStep: Ejecutado después de cada paso.

2.3.7.2 Ejemplo

El código a probar corresponde al fichero “Stack.rb” que contiene la clase “Stack” que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
# Stack - LIFO structure data
  class Stack
    def initialize
        @store = Array.new
    end
    def pop
        @store.pop
    end
    def push (element)
        @store.push (element)
    end
    def size
```

```
            @store.size
        end
    end
```

A continuación, se detallan las pruebas unitarias a realizar sobre la clase Stack:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.
- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.
- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

El usuario define los escenarios en el fichero de características. El contenido del fichero de características es el siguiente (stack.features):

```
# language: En
Features: Stack Management
        As a tester
        I want to test the Stack class ad its methods
        So that I can validate the developped softwared
Scenario: Pop method should return the top element
        Given a stack
        When I push the item "1" in the stack
        And I push the item "2" in the stack
        And I push the item "3" in the stack
        And I push the item "4" in the stack
        And I pop an item from the stack
        Then I should obtain the item "4"
Scenario: Pop method should remove the top element
        Given a stack
        When I push the item "1" in the stack
        And I push the item "2" in the stack
        And I push the item "3" in the stack
        And I push the item "4" in the stack
        And I pop an item from the stack
        Then the stack size is 3
Scenario: Pop method should add one element
        Given a stack
        When I push the item "A" in the stack
        Then the stack size is one
Scenario: Pop method should add the top element
```

```
Given a stack
When I push the item "A" in the stack
And I push the item "B" in the stack
Then "B" is the last item in the stack.
```

El usuario tiene que definir el código fuente con la implementación de las definiciones de pasos. El fichero de definición de pasos es el siguiente (stack_steps.rb):

```
Given /^a stack$/          do
   @stack = Stack.new
end
When /^I push the item "(.*)" in the stack$/ do |arg1|
  @stack.push (arg1)
end
When /^I pop an the item from the stack$/
  @stack.pop
end
Then /^I should obtain the item "(.*)"$/ do |arg1|
   @stack.pop (arg1)
end
Then /^the stack size is 3$/ do
   @stack.size == 3
end
Then /^the stack size is one$/ do
   @stack.size == 1
end
Then /^”(.*)” is the last item in the stack$/ do |arg1|
   @stack.pop == arg1
end
```

2.3.7.3 Limitaciones

Cucumber es único marco de trabajo de los analizados que permite que las pruebas definidas sean entendibles por todas las partes de involucradas en el desarrollo de un producto software, además de proporcionar un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas. No obstante, presenta algunas limitaciones respecto al problema planteado que se detallan en esta sección.

Cucumber separa el proceso de definición de las pruebas en dos partes:

- Definición de los escenarios y características de pruebas, donde se escriben las pruebas a través de un DSL externo basado en el lenguaje natural.
- Implementación del código fuente para automatizar las pruebas, utilizado básicamente la sintaxis de lenguaje Ruby.

La separación del proceso de definición del proceso de codificación implica que el usuario debe realizar, y por tanto conocer, dos procesos separados para la realizar pruebas unitarias. Además, el proceso de codificación se basa en el uso de la sintaxis de Ruby para la codificación de las pruebas, no proporcionando utilidades específicas que permitan validar el comportamiento del software bajo prueba (e.j. no proporciona ni asertos ni expectaciones para facilitar la realización de pruebas). Por lo tanto, podemos concluir que Cucumber no facilita en dicho aspecto el proceso de codificación de pruebas unitarias a los desarrolladores o programadores de pruebas unitarias comparado con otros marcos de trabajos analizados.

2.3.8 Shoulda/Context

En las siguientes secciones se describe el mecanismo que proporciona Shoulda-Context para la definición de pruebas unitarias junto con un pequeño ejemplo, que ayudará a contextualizar y entender el mecanismo de definición de pruebas.

Además, se detallarán las limitaciones de Shoulda-Context respecto al problema planteado en la investigación.

2.3.8.1 Descripción

Should/Context es un DSL interno que permite definir pruebas más legibles y entendibles en Test/Unit y Minitest/Unit. Permite definir el nombre de los métodos de pruebas sin utilizar "_", lo que permite identificar más claramente el objeto de la prueba, y definir contextos de pruebas lo que permite agrupar pruebas acorde una característica específica o escenario determinado, además de proporcionar bloques "should".

Un bloque contexto ("context (name&block)") define un escenario de prueba que agrupa un conjunto de estamentos "should" junto con un conjunto común de bloques "setup" y "teardown", lo que mejora la comprensión y mantenimiento de las pruebas definidas.

Los bloques "setup" y "teardown" permite definir las acciones que se hace antes y después de la ejecución de cada prueba.

Los bloques "should" proporcionan azúcar sintáctica sobre los métodos de pruebas de Test/Unit y Minitest/Unit. Un bloque "should" contiene las aserciones y el código necesario para la prueba, con el beneficio de poder agruparlos dentro de un bloque "contexto".

Bloques Should	Descripción
shoud (name_or_matcher, options={}, &block)	Permite realizar pruebas positivas.
should_not (matcher)	Permite realizar pruebas negativas.
should_eventually (name,&block)	Similar a "should", pero nunca se ejecuta.
subject (&block)	Establece el valor que devuelve el método instanciado.
block: bloque de código name: nombre de la prueba matcher: métodos matcher de Ruby (ej. equal, include, respond_to, etc..) o expresiones regulares.	

Tabla 9 - Bloques Should

Should/Context también proporciona varios asertos adicionales a los marcos de trabajo Test/Unit y Minitest/Unit:

Asertos	Descripción
assert_accepts (pattern, target}	Valora que un valor se ajusta a un patrón dado.
assert_reject (pattern, target}	Valora que un valor no se ajusta a un patrón dado.
assert_same_elements (a1, a2, [msg])	Valora si dos arrays (a1, a2) contienen los mismos elementos. Ej. assert_same_elements ([:a, :b, :c], [:c, :a, :b]).
assert_contains (collection, x)	Valora si un elemento determinado "X" está en el array "collection". Ej. assert_contains (['a', '1'], 'a').
assert_does_not_contain (collection, x)	Valora si un elemento "X" no está en el array "collection". Ej. assert_contains (['a', '1'], 'a').

Tabla 10 - Asertos Adicionales Shoulda/Context

A continuación se presenta un ejemplo de "Test/Unit sin Shouda/Context" y el mismo ejemplo con "Test/Unit junto con Shoulda/Context":

```
require "test/unit"
class CalculatorTest < Test::Unit::TestCase
    def setup
        @calculator = Calculator.new
    end
    def add_two_numbers_for_the_sum
        assert_equal 4, @calculator.sum (2, 2)
    end
    def multiply_two_numbers_for_the product
        assert_equal 10, @calculator.product (2, 5)
    end
end
```

El mismo ejemplo de Test/Unit con Shoulda-Context:

```
require "test/unit"
class CalculatorTest < Test::Unit::TestCase
    context "a calculator" do
        setup do
            @calculator = Calculator.new
```

```
        end
        should "add two numbers for the sum" do
            assert_equal 4, @calculator.sum (2, 2)
        end
        should "multiply two numbers for the product" do
            assert_equal 10, @calculator.product (2, 5)
        end
    end
end
```

2.3.8.2 Ejemplo

El código a probar corresponde al fichero “Stack.rb” que contiene la clase “Stack” que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
# Stack - LIFO structure data
  class Stack
      def initialize
          @store = Array.new
      end
      def pop
          @store.pop
      end
      def push (element)
          @store.push (element)
      end
      def size
          @store.size
      end
  end
```

A continuación, se detallan las pruebas unitarias a realizar sobre la clase Stack:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.
- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.
- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

```
require "shoulda/context"
require "stack.rb"
require "test/unit"
```

```
class TestStack < Test::Unit::TestCase
    context "a stack have two methods to manage: pop and push"
            setup do
                    @stack = Stack.new
            end
            should "pop method obtain the top element" do
                    @stack.push (1)
                    @stack.push (2)
                    @stack.push (3)
                    @stack.push (4)
                    assert_equal (4, @stack.pop)
            end
            should "pop method remove the top element" do
                    @stack.push (1)
                    @stack.push (3)
                    @stack.push (5)
                    @stack.push (7)
                    @stack.pop
                   assert_equal (3, @stack.size)
            end
            should "push method add one element" do
                    @stack.push ("a")
                    assert_equal (1, @stack.size)
            end
            should "push method add the top element" do
                    @stack.push ("a")
                    @stack.push ("b")
                    assert_equal ("b", @stack.pop)
            end
        end
    end
```

2.3.8.3 Limitaciones

Shoulda-Context permite que las pruebas definidas en Unit/Test y Minitest/Unit sean más legibles y comprensibles. No obstante, las pruebas con Should/Context no son legibles ni comprensibles por todas las partes involucradas en el desarrollo del programa (personal técnico, no técnico, etc.) ya que utiliza un lenguaje todavía alejado del lenguaje natural. Debido a ello, no es mecanismo que permita detectar errores de entendimiento en el desarrollo de un producto por cualquier persona involucrada en el desarrollo del mismo durante la fase de pruebas unitarias.

Should-Context no proporciona un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas.

2.3.9 Coulda

En las siguientes secciones se describe el mecanismo que proporciona Coulda para la definición de pruebas unitarias junto con un pequeño ejemplo, que ayudará a contextualizar y entender el mecanismo de definición de pruebas.

Además, se detallarán las limitaciones de Coulda respecto al problema planteado en la investigación.

2.3.9.1 Descripción

Coulda es un DSL interno basado en Test/Unit e inspirado en Cucumber para realizar pruebas unitarias en Ruby.

A modo de recordatorio, Cucumber separa el proceso de descripción de las pruebas (características y escenarios) del proceso de codificación o implementación de las pruebas. Una característica está compuesta por un título (identifica en pocas palabras una actividad para la cual un usuario emplea al sistema), una breve narración (narración para soportar el contexto para los escenarios ejecutables que se tienen que escribir) y por un número arbitrario de escenarios. Los escenarios definen la prueba y están compuestos de pasos que empiezan con “Given”, “When”, “Then”, donde “Given” y “When” definen las acciones de la prueba y “Then” define el resultado de la prueba. Para la descripción de las características y los escenarios de prueba, utiliza un DSL externo basado en el lenguaje cercano al lenguaje natural (lenguaje Gherkin) y para la codificación de los pasos utiliza básicamente la sintaxis del lenguaje Ruby.

La idea de Coulda es unificar el proceso de descripción de los escenarios de prueba y con el proceso de codificación de la prueba. La codificación de la prueba debe incluir la descripción del escenario para el que se está realizando la prueba. Para ello, proporciona un DSL interno, basado en el lenguaje Gherkin, que permite describir los escenarios de prueba y codificar las pruebas a través del DSL, los asertos de Test-Unit y la sintaxis del lenguaje Ruby. La mayoría de los elementos del lenguaje Gherkin (Features, Given, When, Then,..) son definidos como métodos para ser invocados durante la codificación de las pruebas.

Coulda recomienda expresar los escenarios de prueba de la siguiente manera

- Usar el método “Given” para establecer las precondiciones de una prueba.
- Usar el método “When” definir el comportamiento a probar
- Usar el método “Then” para definir el resultado esperado de la prueba. Usar los asertos de Test-Unit para ello.

La siguiente imagen muestra la estructura de pruebas de Coulda formada por el bloque “Features” donde se describe en el objetivo de las pruebas y un bloque “Scenario” correspondiendo cada escenario con una prueba.

```
Feature "Define a feature" do
    in_order_to "write acceptance tests with less code"
    as_a "developer"
    i_want_to "use a simple internal DSL"

  Scenario "Demonstrate how coulda works" do
    Given "a pending prereq" do
    # Precondition code could go here
    end

    When "something happens" do
    # Behavior to test invoked here
    end

    Then "expect something else" do
    # Assertions on result of behavior go here
    end
  end
end
```

Imagen 6 – Estructura Pruebas en Coulda

2.3.9.2 Ejemplo

El código a probar corresponde al fichero “Stack.rb” que contiene la clase “Stack” que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
# Stack - LIFO structure data
  class Stack
      def initialize
            @store = Array.new
      end
      def pop
            @store.pop
      end
      def push (element)
            @store.push (element)
      end
      def size
            @store.size
      end
  end
```

A continuación, se detallan las pruebas unitarias a realizar sobre la clase Stack:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.
- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.

- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

```
require "rubygems"
require "stack.rb"
require "coulda"
include Coulda
Feature "Stack Management" do
    in_order_to "validate the developed software"
    as_a "tester"
    i_want_to "test the Stack class and its methods"
        Scenario "Pop method should return the top element" do
            Given "a stack" do
                @stack = Stack.new
            end
            When "I push the items '1', '2', '3' and '4' in the stack" do
                @stack.push (1)
                @stack.push (2)
                @stack.push (3)
                @stack.push (4)
            end
            Then "I should obtain item '4' using pop method" do
                assert_equal (4, @stack.pop)
            end
        end
        Scenario "Pop method should remove the top element" do
            Given "a stack" do
              @stack = Stack.new
            end
            When "I push the items '1', '2', '3' and '4' in the stack" do
                @stack.push (1)
                @stack.push (3)
                @stack.push (5)
                @stack.push (7)
            end
            And "I pop an item from the stack" do
                @stack.pop
            Then "the stack size must be 3" do
                assert_equal (3, @stack.size)
```

```
        end
    end
    Scenario "Push should add one element" do
        Given "a stack" do
            @stack= Stack.new
        end
        When "I push the item 'a' in the stack" do
            @stack.push (“a”)
        end
        Then "the size of the stack is 1" do
            assert_equal (1, @stack.size)
        end
    end
    Scenario "Push should add the top element" do
        Given "a stack" do
            @stack= Stack.new
        end
        When "I push the item '1' and "2" in the stack" do
            @stack.push (1)
            @stack.push (2)
        end
        Then "2 is the last item of the stack" do
            assert_equal (2, @stack.pop)
        end
    end
end
```

2.3.9.3 Limitaciones

Las pruebas definidas con Coulda son mucho más legibles y comprensibles que las pruebas definidas en otros marcos de trabajo y con otros DSLs. Coulda permite detectar errores de entendimiento en el desarrollo de un producto por cualquier persona técnica involucrada en el desarrollo del mismo durante la fase de pruebas unitarias, gracias a las descripciones usadas en las invocaciones a los métodos (Feature, in_order_to, Scenario, Given, When, Then, etc.) proporcionados con Coulda. No obstante, puede no ser útil para personal no técnico, ya que requiere unos conocimientos mínimos de lenguaje Ruby para entender las pruebas definidas con Coulda.

El uso de los asertos de Unit/Test para valorar si una prueba pasa o falla, hace que Coulda se aleje de la idea de utilizar el lenguaje natural para expresar y codificar el objeto de la prueba. El uso de un lenguaje similar a las expectaciones (e.j. "must", should_be_equal, etc.) en lugar de asertos acercaría Coulda al ideal uso del lenguaje natural para codificar pruebas unitarias.

Coulda no proporciona un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas.

2.3.10 Shindo

En las siguientes secciones se describe el mecanismo que proporciona Shindo para la definición de pruebas unitarias junto con un pequeño ejemplo, que ayudará a contextualizar y entender el lenguaje que proporciona para la definición de pruebas.

Además, se detallarán las limitaciones de Shindo respecto al problema planteado en la investigación.

2.3.10.1 Descripción

Shindo es un marco de trabajo muy simple, basado en un concepto similar a los asertos, para la realización de pruebas unitarias sobre trozos de código siempre referidos a objetos y no al comportamiento global de un sistema.

Las pruebas en Shindo se basan en dos conceptos:

- Una prueba proporciona un resultado, o
- Una prueba genera un error.

Shindo proporciona el método “tests” para definir una prueba y los métodos “returns” y “raises” para valorar si el resultado de la prueba es el esperado. El método “tests (description = “returns true”, &block)” toma como argumentos una descripción de la prueba y un bloque de código correspondiente la prueba a realizar. Dentro del bloque de código, se incluyen los asertos (“returns” y “raises”) que permiten valorar si la prueba pasa o falla.

El método “returns (expectation, description = “returns#{expectation.inspect}”, &block)” chequea si el valor esperado (“expectation”) coincide con el valor devuelto tras la ejecución del bloque de código (“&block”).

El método "raises (error, description = “raise#{error.inspect}”,&block)" chequea que el error esperado (“error”) coincide con el error generado tras la de la ejecución del bloque de código (“&block”).

También proporciona los métodos “after” y “before” para establecer el contexto de la prueba. El método “after (&block)” se ejecuta antes de cada prueba y el método before (&block) se ejecuta después de cada prueba.

Shindo permite el anidamiento de pruebas dentro de los bloques código de otras pruebas.

2.3.10.2 Ejemplo

El código a probar corresponde al fichero “Stack.rb” que contiene la clase “Stack” que dispone de tres métodos (pop, push y size):

```
# The Book of Ruby - http://www.sapphiresteel.com
 # Stack - LIFO structure data
   class Stack
       def initialize
              @store = Array.new
       end
       def pop
              @store.pop
```

```
    end
    def push (element)
        @store.push (element)
    end
    def size
        @store.size
    end
end
```

A continuación, se detallan las pruebas unitarias a realizar sobre la clase Stack:

- Prueba 1: Apilar varios elementos y usar el método “pop” para obtener el último elemento apilado.
- Prueba 2: Apilar varios elementos y usar el método “pop” para sacar el último elemento apilado y determinar cuántos elementos quedan en la pila.
- Prueba 3: Apilar un elemento mediante el método “push” y determinar cuántos elementos hay la pila.
- Prueba 4: Apilar varios elementos mediante el método “push” y determinar que el método “push” incluye elementos por el final.

```
require "stack.rb"
require "shindo"
Shindo.tests ('stack management') do
    before do
        @stack = Stack.new
    end
    tests ("pop returns the top element") do
        @stack.push ("a")
        @stack.push ("b")
        @stack.push ("c")
        @stack.push ("d")
        returns ("d") {@stack.pop}
    end
    tests ("pop removes the top element") do
        @stack.push ("1")
        @stack.push ("3")
        @stack.push ("5")
        @stack.push ("7")
        @stack.pop
        returns (3) {@stack.size}
    end
    tests ("push method add one element") do
        @stack.push ("a")
```

```
        returns (1) {@stack.size}
    end
    tests ("push method add the top element") do
        @stack.push ("a")
        @stack.push ("b")
        returns ("b") {@stack.pop}
    end
end
```

2.3.10.3 Limitaciones

Shindo proporcionan un mecanismo ágil y rápido para la realización de pruebas unitarias, pero con el inconveniente que la forma de escribir las pruebas unitarias se aleja bastante del lenguaje natural.

Las pruebas con Shindo no son legibles ni comprensible por todas las partes involucradas en el desarrollo del programa (personal técnico, no técnico, ..) ya utiliza un lenguaje todavía alejado del lenguaje natural. Debido a ello, no es mecanismo que permita detectar errores de entendimiento en el desarrollo de un producto por cualquier persona involucrada en el desarrollo del mismo durante la fase de pruebas unitarias.

Shindo no proporciona un mecanismo para formalizar las pruebas realizadas, de manera que estas puedan servir para la aceptación del código desarrollado por las partes involucradas.

3. COMO EVALUAR LAS SOLUCIONES

En esta sección se detallaran cuáles son los factores relevantes para evaluar los marcos de trabajo de pruebas unitarias de Ruby frente al problema detectado en este TFM y las bondades del prototipo desarrollado en este TFM que intenta suplir algunas de las deficiencias de los marcos de trabajo de pruebas unitarias evaluados.

Para poder evaluar los diferentes marcos de trabajo y las bondades del prototipo frente a otros marcos de trabajo para pruebas unitarias, es necesario conocer cómo funcionan dichos marcos de trabajo de pruebas unitarias en Ruby. En la sección anterior, ya se ha detallado el funcionamiento de los marcos de trabajo de pruebas unitarias para Ruby más populares, incluyendo otras herramientas cuyo objetivo no es la realización de pruebas unitarias (caso de Cucumber), pero que se pueden utilizar para tal fin. También, se ha incluido un ejemplo de pruebas que permite entender con mayor facilidad cuales son las labores del usuario a la hora desarrollar las pruebas unitarias, la interfaz proporcionada para la definición de las pruebas, la complejidad del proceso de definición de las pruebas, el tiempo requerido para entender el proceso de definición de las pruebas, la legibilidad de las pruebas definidas y el resultado final del proceso.

Así pues, los factores que se deben tener cuenta para evaluar las distintas soluciones (las existentes en el mercado y el prototipo desarrollado en este TFM) para resolver el problema por la que surge este TFM son los siguientes:

- Legibilidad, claridad y comprensión de las pruebas realizadas por todo el personal involucrado en el desarrollo del software.
- Mecanismo de documentación formal de las pruebas realizadas.

- Codificación de pruebas unitarias con un lenguaje cercano al natural.
- Separación del proceso de definición de pruebas del proceso de codificación de pruebas.

A la hora de evaluar las soluciones, debemos dar respuesta a las siguientes preguntas:

- ¿Las pruebas definidas son legibles y comprensibles por el personal técnico, el no técnico y por el cliente?
- ¿La solución proporciona un mecanismo de documentación formal de las pruebas realizadas y que puedan ser utilizadas para la aceptación del software desarrollado?
- ¿Permite la codificación de pruebas usando un lenguaje propio de pruebas y cercano al lenguaje natural?
- ¿Existe separación entre el proceso de definición de las pruebas y el proceso de desarrollo del código fuente necesario para la automatización de dichas pruebas?

4. DESCRIPCIÓN DETALLADA DE LA SOLUCIÓN PROPUESTA

La solución plantea un marco de trabajo (Tests_Tool en adelante) para la realización de pruebas unitarias en Ruby que permite realizar pruebas unitarias entendibles y legibles por las personas involucradas en el desarrollo un producto software, sin la necesidad de conocer el lenguaje Ruby ni la necesidad de tener conocimientos específicos del marco de trabajo con el que se realizan las pruebas unitarias. Para ello, proporciona un DSL interno cercano al lenguaje natural para definir y codificar las pruebas, y genera automáticamente un fichero de texto con las pruebas definidas estando estas descritas en el lenguaje natural.

La solución también plantea un marco de trabajo para la realización de pruebas unitarias en Ruby que facilita el proceso de codificación de las pruebas unitarias al programador. Para ello, proporciona un DSL interno basado en el lenguaje natural que facilita el proceso de codificación de las pruebas unitarias y un conjunto de expectaciones que permiten evaluar el comportamiento del software desarrollado.

La solución también plantea un marco de trabajo para la realización de pruebas unitarias en Ruby que permite utilizar las pruebas unitarias definidas como pruebas de aceptación y validación del software desarrollado por las partes involucradas en el proceso de desarrollo de un producto software. Para ello, proporciona un DSL interno basado en el lenguaje natural que permite entender de forma clara y legible las pruebas realizadas, una interfaz de salida de datos que proporciona las pruebas realizadas y el resultado de las mismas y genera un fichero de texto con las pruebas realizadas y el resultado de las mismas. Es importante resaltar, que la interfaz de salida de datos con los resultados de la ejecución y el fichero de texto, describen las pruebas ejecutadas en lenguaje natural.

4.1 Descripción de Alto Nivel de la Solución

La definición y codificación de las pruebas unitarias se realizada a través de un editor de texto plano o editor de código de fuente que permitan editar ficheros con extensión “rb”. El usuario utilizará el lenguaje de dominio específico proporcionado por el marco de trabajo Tests_Tool para definir y codificar las pruebas unitarias en un fichero con extensión “rb”.

La solución proporciona un formato estándar para definir pruebas en forma de Pasos de Prueba, Casos de Prueba y Suite de Pruebas:

- Paso de Prueba es un ejemplo ejecutable de como el software bajo prueba puede ser usado y su comportamiento esperado en un contexto determinado.
- Caso de Prueba es un conjunto de pasos de pruebas agrupadas bajo un determinado criterio, por ejemplo, conjunto de pruebas que verifican una característica concreta del producto desarrollado. La agrupación de pasos de prueba en casos de prueba es opcional, aunque se recomienda agrupar los pasos de prueba en casos de prueba.
- Suite de Pruebas es un conjunto de casos de prueba agrupados bajo un determinado criterio. La agrupación de casos de prueba en suite de pruebas es opcional.

El marco de trabajo proporciona un lenguaje de dominio específico interno al lenguaje de programación Ruby (metalenguaje), formado por un conjunto reducido de palabras claves y un conjunto de expectaciones que permiten evaluar el comportamiento del software desarrollado, que junto con la sintaxis y utilidades proporcionadas por el lenguaje Ruby permiten la realización de pruebas unitarias.

Las palabras claves del lenguaje de dominio específico con las siguientes:

- Test _Step
- Test _Case
- Tests_Suite
- Given
- When
- Then
- And
- The_purpose_is
- It_is_important
- In_addition
- In_order_to
- I_want_to
- As_a
- For_that
- The
- Even
- Although

Las expectaciones que proporciona el marco de trabajo se forman mediante la palabra clave “should”, y adicionalmente junto con su combinación con las siguientes palabras claves:

- be
- not
- satisfy
- equal
- identical_to
- higher_than
- lower_than
- close
- raise
- match

Todas estas palabras clave y las expectaciones son métodos que requieren como argumento una descripción y, la mayoría de ellos, un bloque de código contiendo las acciones a realizar para automatizar las pruebas unitarias.

Una vez las pruebas unitarias han sido definidas y codificadas en el fichero con extensión “rb” por el usuario, el fichero tiene que ser ejecutado manualmente y como resultado de su ejecución el marco de trabajo Tests_Tool proporciona una salida de datos en la consola del sistema operativo con la descripción de las pruebas ejecutadas y su resultado. También, genera un fichero “txt” donde se describen las pruebas ejecutadas y el resultado de la ejecución.

La siguiente imagen (Imagen 7) muestra la arquitectura de la solución, en la cual se identifican los elementos que forman parte de ella:

- Fichero de pruebas (ej. tests.rb) donde el usuario define las pruebas utilizando el DSL interno (metalenguaje en Ruby) proporcionado por la herramienta Test_Tool.

- Fichero Tests_Tool.rb (herramienta Tests_Tool) que proporciona el DSL para definición de pruebas unitarias, así como funciones de soporte para la presentación de resultados y la generación del fichero de "txt" con las evidencias de las pruebas ejecutadas.
- Librerias de Ruby (RubyGems) usadas por la herramienta Tests_Tool.
- Fichero (ej.CodigoFuente.rb) con el código fuente objeto de prueba.
- Compilador de Ruby encargado de analizar las pruebas definidas en el fichero de pruebas mediante el DSL y convierte las pruebas definidas en código ejecutable, utilizando los ficheros de la herramienta Test_Tool.rb, las librerías de Ruby y el código fuente bajo a prueba. Como resultado de la ejecucíon del ejecutable, se genera el fichero "txt" con las evidencias de las pruebas ejecutadas y se presentan en la consola el resultado de la ejecución de las pruebas.

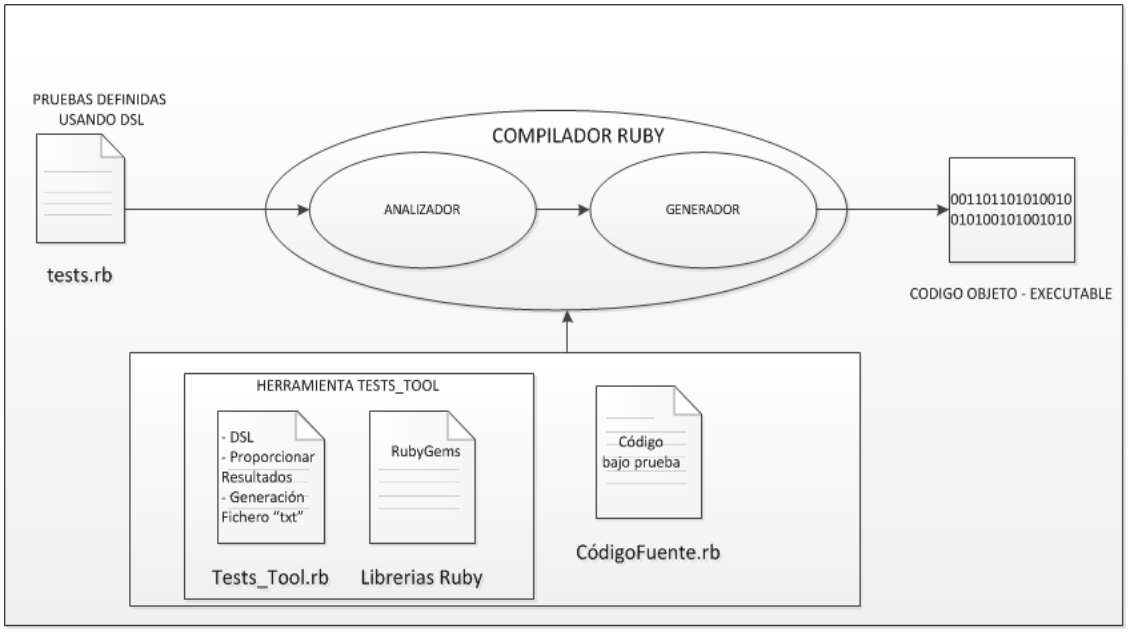


Imagen 7 – Arquitectura de la Solución

4.2 Descripción Funcional de la Solución y del Prototipo

Antes de profundizar en los detalles funcionales de solución es necesario conocer que es una historia de usuario ya que es la base para entender la solución propuesta en este marco de trabajo para la definición de pruebas unitarias. Una historia de usuario es una representación de un requisito escrito en varias frases utilizando el lenguaje común del usuario y utilizada en las metodologías ágiles para la especificación de requisitos (acompañada de las conversaciones con los usuarios y unos criterios de aceptación que permiten validar el comportamiento del sistema). Las Historias de Usuario sirven para describir lo que el usuario desea ser capaz de hacer. Además, las historias de los usuarios se centran en el valor que viene de usar el sistema en lugar de una especificación detallada de lo que el sistema debe hacer. Están concebidos como un medio para fomentar la colaboración.

Una historia de usuario consta de:

- Un título que describe brevemente una actividad.
- Narrativa que sirve para aclarar los detalles del requisito o funcionalidad.

- Criterio/s de aceptación que permita determinar cuándo la historia ha sido completada.

No existe una plantilla estándar para la definición de historias de usuario. No obstante, la mayoría de historias siguen un formato muy parecido para la definición de las mismas. Para la definición del título y la narrativa se suele utilizar el siguiente formato:

Título (una línea describiendo la historia)

Narrativa:

Yo como (As a) <rol>

Necesito/deseo/quiero (I want to) <funcionalidad o característica>

Para (in order to) <beneficio de negocio>

Para la definición de los criterios de aceptación se pueden manejar dos opciones:

- Criterios de Aceptación en Prosa, donde se describe los criterios de aceptación.
- Criterios de Aceptación en Formato BDD.
    - GIVEN - DADO
    - WHEN - CUANDO
    - THEN - ENTONCES

A continuación se presenta un ejemplo de una historia de usuario “Ejemplo: Ingreso al Sistema”:

Historia Usuario – Entrar al Sistema

Yo como (As a) Usuario

Deseo (I want to) acceso al sistema

Para (In order to) poder uso de las funcionalidades del sistema.

---- Criterios de Aceptación ---

Criterio Nº1: Acceso Exitoso

- Dado (Given) que se han rellenado correctamente la contraseña.
- Cuando (When) el usuario pulse botón “Aceptar”.
- Entonces (Then) se permitirá el acceso al sistema.

Criterio Nº2: Acceso Fallido

- Dado (Given) que se han rellenado la contraseña
- Cuando (When) el usuario pulse botón “Aceptar”.
- Entonces (Then) no se permitirá el acceso al sistema.

El lenguaje de dominio específico proporcionado en este marco de trabajo para la realización de pruebas unitarias se basa en los principios de las historias de usuario. El objetivo es escribir pruebas como si fuesen una historia de usuario, ampliando los elementos de son parte de la narrativa de una historia de usuario.

Se puede establecer una relación entre una historia de usuario y los elementos del lenguaje de dominio definido en este TFM:

- Una Suite de prueba se puede equiparar a una historia de usuario, con un título y una narrativa que describe cual es el objetivo de un conjunto de

casos de pruebas para validar una determina funcionalidad o característica del sistema.

- Un caso de prueba se puede equiparar a una historia de usuario, con un título y una narrativa que describe cual es el objetivo de las pruebas para validar una determina funcionalidad o característica del sistema.
- Un paso de prueba sería equiparable al concepto de criterio de aceptación de una historia de usuario. Un paso de prueba se define mediante la fórmula Given/When/Then.

En las siguientes secciones se detalla el lenguaje de dominio específico proporcionado por el marco de trabajo, la interfaz de resultados de la ejecución, así como los detalles de implementación del marco de trabajo proporcionado.

4.2.1 Lenguaje de Dominio Específico – Test_Step

Un paso de prueba es un ejemplo concreto de cómo el sistema bajo prueba debe comportarse. Un paso de prueba se define mediante el método Test_Step que requiere como argumentos un texto que identifica el paso de prueba y un bloque de código formado por los siguientes elementos:

- “Given”. El método “Given” se usa para definir el contexto o las precondiciones para llevar a cabo la prueba. El método requiere como parámetros de entrada una breve descripción del contexto y un bloque de código que define las acciones a realizar para obtener contexto necesario para la prueba.
- “When”. El método “When” se usa para definir el evento o la acción que ocurre en dentro del contexto definido y que desencadena el comportamiento que está bajo prueba. El método requiere como parámetros de entrada una breve descripción y un bloque de código donde se define la acción o evento que desencadena la prueba.
- “Then”. El método “Then” se usa para definir el resultado esperado de la prueba. El método requiere como parámetros de entrada una breve descripción y un bloque de código donde se define las expectaciones (una expectación es una expresión de cómo el código se espera que se comporte) que permitirán evaluar el resultado obtenido de la prueba. El paso de prueba se da por pasado cuando el resultado obtenido coincide con el resultado esperado, y falla, cuando el resultado obtenido no es el esperado.

Es importante recordar que todo bloque de código está delimitado en Ruby mediante las palabras clave do/end (ej. do block end) o mediante { } (ej. {block}).

A continuación se detalla la estructura requerida para un paso de prueba:

```
Test_Step "nombre de la prueba" do
        Given "Precondiciones" do
            #bloque de código donde se definen las precondiciones
        end
        When "Algo ocurre" do
            #bloque de código de la acción o evento
            #que permite verificar el comportamiento bajo prueba
        end
```

```
        Then "El comportamiento esperado es" do
            #Bloque de código que permite evaluar que el resultado
            #obtenido es el esperado.
           #Usar las expectaciones para ello.
        end
    end
```

Adicionalmente, un paso de prueba permite usar “And” tras “Given” y “When”.

```
    Test_Step "nombre" do
        Given "Precondición1" do
           #bloque de código
        end
        And "Precondición2" do
           #bloque de código
        end
        When "ocurre el evento 1" do
           #bloque de código
        end
        And "ocurre el evento 2" do
           #bloque de código
        end
        Then "El comportamiento esperado es" do
           #bloque de código
           #Expectaciones.
        end
    end
```

Es importante resaltar que el usuario del marco de trabajo debe dar sentido a las descripciones utilizadas como argumento en los métodos Given, When, And y Then para que los pasos de prueba sean entendibles y legibles.

La fórmula Given/When/Then guía la definición de pruebas de unitarias basados en una historia de usuario y nos permite definir pruebas de manera fácil, a la vez entendibles y legibles. El uso de Given/Then/When para definir pruebas unitarias nos permite:

- Comunicar el propósito de la prueba claramente al dividir los elementos de una prueba en tres secciones diferenciadas: precondición o estado inicial del sistema para la realizar la prueba, la acción o evento que dispara la prueba y la postcondición o resultado esperado de la prueba refleja el estado del sistema una vez ejecutada la prueba.
- Centrar al usuario en lo que está haciendo mientras define las pruebas, ya que al relacionar un nombre (ej. Given = precondición) con una sección de la prueba hace más fácil saber lo que se está escribiendo

- Hacer pruebas de una manera más rápida al tener un formato determinado y siempre el mismo para la definición de pruebas.
- Hace más fácil el reúso de partes de la prueba al poder reutilizar secciones enteras de una prueba en otras (ej. El bloque Given de una prueba se puede reutilizar en otras).
- Resaltar las asunciones que se están haciendo sobre las precondiciones de la prueba al tener una sección específica para ella en la prueba (sección Given).
- Resaltar la acción (o evento) que dispara la prueba al tener una sección específica para ella en la prueba (sección When).
- Resaltar el resultado esperado en la prueba que se está realizando al tener una sección específica para ella en la prueba (sección Then).

Es importante resaltar que se podrían haber utilizado otro tipo de estructuras menos estandarizadas para la definición de los pasos de prueba como “Precondicions/Trigger/Postcondition” o “Setup/Exercise/Verify”, siendo estas igual de exitosas que la fórmula “Given/When/Then”.

4.2.2 Lenguaje de Dominio Específico – Test_Case

Un caso de prueba es una agrupación de pasos de prueba, agrupados bajo el criterio del usuario y que normalmente agrupan pruebas relacionadas con una funcionalidad o característica del sistema a verificar. Un caso de prueba se define mediante el método Test_Case que requiere como argumentos un texto que identifica el caso de prueba y un bloque de código que contiene los métodos descriptores del propósito del caso de prueba y los pasos de prueba que forman parte del caso prueba. Para describir el propósito de prueba, se proporcionan los siguientes métodos que pueden ser usados de manera individual o varios a la vez en un mismo caso de prueba:

- In_order_to. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- The_purpose_is. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- It_is_important. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- In_addition. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- I_want_to. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- As_a. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- For_that. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.

- The. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- Although. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- Even. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.

Es importante resaltar que es el usuario del marco de trabajo debe dar sentido a las descripciones utilizadas como argumento en los métodos citados (In_order_to, The_purpose_is, etc.) y al uso de uno o varios de los citados métodos para que la descripción del propósito de cada caso de prueba sean entendibles y legibles.

A continuación se detalla un ejemplo con la estructura requerida para un caso de prueba:

```
Test_Case “Nombre” do
    The_purpose_is “descripción”
    Its_important_to “descripción”
    In_order_to “descripción”
    As_a “descripción”
    I_want_to “descripción”
    Test_Step "Nº1 nombre de la prueba" do
        Given "Precondiciones" do
            #bloque de código
        end
        When "Algo ocurre" do
            #bloque de código
        end
        Then "El comportamiento esperado es" do
            #Bloque de código
        end
    end
    Test_Step "Nº2 nombre de la prueba" do
        Given "Precondiciones" do
            #bloque de código
        end
        When "Algo ocurre" do
            #bloque de código
        end
        Then "El comportamiento esperado es" do
            #Bloque de código
        end
```

```
        end
    end
```

Es importante resaltar que la organización de pasos de prueba en casos de pruebas es opcional, es decir, el marco de trabajo está diseñado tanto para que los casos de pruebas se agrupen o no en casos de pruebas. No obstante, se recomienda agrupar los pasos de prueba en casos de prueba ya que facilita la comprensión de las pruebas realizadas.

En caso de usar casos de pruebas para agrupar pruebas, un caso de prueba debe contener al menos un paso de prueba.

4.2.3 Lenguaje de Dominio Específico – Tests_Suite

Una suite de pruebas es una agrupación de casos de prueba, agrupados bajo el criterio del usuario. Una suite de pruebas se define mediante el método Tests_Suite que requiere como argumentos un texto que identifica la suite de pruebas y un bloque de código que contiene los métodos descriptores del propósito de la suite de prueba y los casos de prueba que forman parte de la suite de pruebas. Para describir el propósito de prueba, se proporcionan los siguientes métodos que pueden ser usados de manera individual o varios a la vez en un mismo caso de prueba:

- In_order_to. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- The_purpose_is. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- It_is_important. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- In_addition. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- I_want_to. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- As_a. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- For_that. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- The. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.
- Although. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.

- Even. Permite describir el propósito de las pruebas. El método requiere como argumento de entrada un texto usado para explicar el propósito de las pruebas.

Es importante resaltar que es el usuario del marco de trabajo debe dar sentido a las descripciones utilizadas como argumento en los métodos citados (In_order_to, The_purpose_is, etc.) y al uso de uno o varios de los citados métodos para que la descripción del propósito de cada suite de pruebas sean entendibles y legibles.

A continuación se detalla un ejemplo con la estructura requerida para una suite de prueba:

```
Tests_Suite “Nombre” do
    In_order_to “descripción”
    As_a “descripción”
    I_want_to “descripción”
    Even “descripción”
    For_that “descripción”
    The “descripción”
    Although “descripción”
    Test_Case "Nombre" do
        The_purpose_is “descripción”
        Its_important_to “descripción”
        Test_Step “nombre” do
          #bloque Given/When/Then
        End
        Test_Step “nombre” do
          #bloque Given/When/Then
        end
    end
    Test_Case "Nombre" do
        The_purpose_is “descripción”
        Test_Step “nombre” do
          #bloque Given/When/Then
        End
        Test_Step “nombre” do
          #bloque Given/When/Then
        end
    end
end
```

Es importante resaltar que la organización de casos de prueba en suite de pruebas es opcional, es decir, el marco de trabajo está diseñado tanto para que los casos de pruebas se agrupen o no en suite de pruebas. No obstante, se recomienda organizar los casos de prueba en suite de prueba siempre que sea necesario para dar claridad a las pruebas realizadas.

4.2.4 Lenguaje de Dominio Específico – Expectaciones

En el bloque de código del método “Then” de un paso de prueba se definen las acciones que nos permiten valorar si la prueba realizada se considera pasada (Passed) o falla (Failed). Un paso de prueba se considera pasado si el comportamiento del software es acorde al esperado en la prueba. Un paso de prueba se considera fallado cuando el comportamiento del software bajo prueba no se ajusta al comportamiento esperado.

El marco de trabajo proporciona un conjunto de expectaciones que permiten valorar si el comportamiento de software está acorde o no al comportamiento esperado del mismo. Una expectación es un método cuyo código permite evaluar el comportamiento del software.

Las expectaciones proporcionadas por el marco de trabajo se forman mediante el método “should”, que requiere un bloque predicado como parámetro de entrada (bloque de código que como resultado produce un valor booleano: true|false). El método se puede combinar con los siguientes métodos para crear macros de expectaciones (expectaciones más complejas):

- be. El método requiere, opcionalmente, un bloque predicado como parámetro de entrada. La prueba se considera pasada si el resultado de evaluar el bloque de código respecto al valor proporcionado por el software bajo prueba devuelve el booleano “true”.
- not. El método requiere, opcionalmente, un bloque predicado como parámetro de entrada. La prueba se considera pasada si el resultado de evaluar el bloque de código respecto al valor proporcionado por el software bajo prueba no devuelve el booleano “true”.
- satisfy. El método requiere como parámetro de entrada un bloque predicado. La prueba se considera pasada si el resultado de evaluar el bloque de código respecto al valor proporcionado por el software bajo prueba devuelve el booleano “true”.
- match. El método requiere un patrón como parámetro de entrada. La prueba se considera pasada si el texto proporcionado por el software bajo prueba se ajusta al patrón recibido como parámetro de entrada.
- raise. El método requiere como argumento de entrada el nombre de una excepción de Ruby y un bloque de código que contiene las acciones para que se produzca la excepción recibida por parámetro. La prueba se considera pasada si el software bajo prueba genera la excepción recibida como parámetro de entrada.
- equal. El método requiere como argumento de entrada un valor que se usa para comparar por el valor producido por el software bajo prueba. La prueba se considera pasada si el valor proporcionado por el software bajo prueba es igual al valor esperado.
- identical_to. El método requiere como argumento de entrada un valor que se usa para comparar por el valor producido por el software bajo prueba. La prueba se considera pasada si el valor proporcionado por el software bajo prueba es igual al valor esperado.
- higher_than. El método requiere como argumento de entrada un valor que se usa para comparar por el valor producido por el software bajo prueba. La prueba se considera pasada si el valor proporcionado por el software bajo prueba es mayor al valor esperado.

- lower_than. El método requiere como argumento de entrada un valor que se usa para comparar por el valor producido por el software bajo prueba. La prueba se considera pasada si el valor proporcionado por el software bajo prueba es menor al valor esperado.
- close. El método requiere como argumento de entrada dos valores numéricos, siendo un valor delta y el otro un valor numérico para definir un rango. La prueba se considera pasada si el valor numérico proporcionado por el software bajo prueba está en un delta del valor número recibido como parámetro de entrada.

Es importante resaltar que los métodos citados siempre tienen que usarse en combinación con el método “should” y nunca sin “should” para el correcto funcionamiento del marco de trabajo.

En la siguiente tabla se detallan las expectaciones que se pueden definir mediante la combinación de los elementos mencionados:

Expectación	Notas
actual.should (predicado)	Pasa si se obtiene el booleano “true” como resultado de evaluar el predicado respecto al valor actual de un objeto. Ej. (2+2).should > 3
actual.should.not (predicado)	Pasa si el booleano “True” se obtiene como resultado de evaluar el predicado respecto al valor actual de un objeto. Ej. (4*2).should.not < 3
actual.should.be (predicado)	Pasa si el booleano “True” se obtiene como resultado de evaluar el predicado respecto al valor actual de un objeto. Ej. (2+2).should.be > 3
actual.should.not.be (predicado)	Pasa si el booleano “True” se obtiene como resultado de evaluar el predicado respecto al valor actual de un objeto. Ej. 4.should.not.be < 3
actual.should.equal (valor esperado)	Pasa si el valor actual de un objeto es igual al valor esperado. Ej. [1,2,3].should.equal [1,2,3]

actual.should.be.equal (valor esperado)	Pasa si el valor actual de un objeto es igual al valor esperado. Ej. 4.should.be.equal 4
actual.should.not.equal (valor esperado)	Pasa si el valor actual de un objeto no es igual al valor esperado. Ej. 5.should.not.equal 4
actual.should.not.be.equal (valor esperado)	Pasa si el valor actual de un objeto no es igual al valor esperado. Ej. [1,2,3].should.not.be.equal [1,4,5]
actual.should.be.identical_to (valor esperado)	Pasa si el valor actual de un objeto es igual al valor esperado. Ej. 4.should.be.identical_to 4
actual.should.not.be.identitical_to (valor esperado)	Pasa si el valor actual de un objeto no es igual al valor esperado. Ej. 5.should.not.be.identical_to 4
actual.should.be.higher_than (valor esperado)	Pasa si el valor actual de un objeto es mayor al valor esperado. Ej. (5+9).should.be.higher_than (4*2)
actual.should.not.be.higher_than (valor esperado)	Pasa si el valor actual de un objeto no es mayor al valor esperado. Ej. (5+9).should.not.be.higher_than (5*9)
actual.should.be.lower_than (valor esperado)	Pasa si el valor actual de un objeto es menor al valor esperado. Ej. (5+9).should.be.lower_than 200
actual.should.not.be.lower_than (valor esperado)	Pasa si el valor actual de un objeto no es menor al valor esperado. Ej. (5*9).should.not.be.lower_than (5+9)

actual.should.be.close (valor límite, delta)	Pasa si el valor actual de un objeto está dentro de un rango definido por un valor límite y un delta. Ej. 4.should.be.close (3, 1)
actual.should.not.be.close (valor límite, delta)	Pasa si el valor actual de un objeto no está dentro de un rango definido por un valor límite y un delta. Ej. 4.should.not.be.close (6, 1)
actual.should.match (patrón)	Pasa si el valor del objeto actual se ajusta al patrón definido mediante una expresión regular. Ej. “tigreton”.should.match (/tigre/)
actual.should.not.match (patrón)	Pasa si el valor del objeto actual no se ajusta al patrón definido mediante una expresión regular. Ej. “tigreton”.should.not.match (/leon/)
proc.should.raise (excepción)	Pasa si el bloque de código “{ }” se genera la excepción definida como parámetro. Ej. lambda {10/0}.should.raise (ZeroDivisionError)
proc.should.not.raise (excepción)	Pasa el bloque de código “{ }” no se genera la excepción definida como parámetro. Ej. Lambda {10/1}.should.not.raise (ZeroDivisionError)
should.satisfy { }	Pasa si el bloque de código genera el valor booleano “true”. Ej. should.satisfy {5 < 10}
should.not.satisfy { }	Pasa si el bloque de código no genera el valor booleano “true”. Ej. should.not.satisfy {5 > 10}
Notas: "actual" es el estado actual del objeto. "esperado" es el estado esperado del objeto. “proc” es un objeto de tipo “Proc” (i.e., es un bloque de código almacenado en un variable).	

Tabla 11 - Expectaciones Prototipo

Se recomienda usar solamente una expectación (simple o compleja) por paso de prueba. La idea es que un paso de prueba sea fácil de escribir y mantener limitando el alcance de cada paso.

Los métodos proporcionados por el lenguaje Ruby son también usados para la construcción de expectaciones. A continuación se muestran algunos de los métodos de Ruby útiles para la construcción de expectaciones:

Método	Notas
Métodos para la gestión de objetos “Array”	
empty?	Devuelve el valor “true” si el array está vacío Ej. a.should.be.empty?
first	Devuelve el primer elemento de un array. “nil” si el array está vacio. Ej. array.first.should.be.equal “a”
last	Devuelve el último elemento de un array. “nil” si el array está vacio. Ej. array.first.last.be.equal “a”
length	Devuelve el número de elementos que tiene el array. Ej. stack.length.should.be.equal 5
include?	Devuelve “true” si un elemento es parte del array. Ej. should.satisfy{ include? (elemento, array)}
&	Devuelve el array resultante a la intersección de dos arrays. ej.[1,2,3] & [2,3,4].should.be.equal [2,3]
+	Devueve el array resultante de la concatenación de arrays. ej.[1,2] + [3,4].should.be.equal [1,2,3,4]
-	Devueve un array con elementos borrados ej.[1,2,3] - [3].should.be.equal [1,2,]
==	Devuelve “true” si dos arrays son iguales Ej. .[1,2,3] - [3].should.be == [1,2,]
Métodos para la gestión de objetos “Numbers” (Integer, floats)	
+	Suma Ej. 2+3.should.be.equal 5
-	Resta Ej. 3-1.should.be.equal 2

*	Multiplicación Ej. 3*2.should.be.equal 6
/	División Ej. 10/2.should.be.equal 5
**	Potencias Ej. 5**2.should.be.equal 25
==, >, <, >=, <=, !=	Igualdad, mayor, mayor igual, menor igual, no igual a. Ej. 4.should.be >=2 34.should.be<54
Métodos para la gestión de objetos “Enumerable”	
all?	Invoca la ejecución del bloque de código suministrado para elemento de la una colección. Devuelve “true” o “false” dependiendo si para cada elemento de la colección, cada llamada al bloque de código devuelve “true”.
any?	Invoca la ejecución del bloque de código suministrado para elemento de la una colección. Devuelve “true” o “false” dependiendo si para alguno de los elementos de la colección, la llamada al bloque de código devuelve “true”.
include?	Devuelve “true” si un elemento es parte de la colección. Ej. should.satisfy { include? (elemento, colección)}
min	Devuelve el menor elemento de la colección.
max	Devuelve el mayor elemento de la colección.
Métodos para la gestión de cualquier objeto	
kind_of?	Devuelve “true” si el objeto es de una clase suministrada. Ej.10.should.be.a.kind_of?(integer)
nil?	Devuelve “true” si el objeto es “nil” Ej. nothing.should.be.nil?
eql? o equal?	Devuelve “true” si un objeto es igual a otro. Ej. texto.should.equal? (texto)
between?	Devuelve “true” si un objeto está entre dos valores o es igual a alguno de ellos. Ej. 4.shoud.be.between? (1, 5).

instance_of?	Devuelve “true” si un objeto es una instancia de una clase determinada. Ej. 4.should.be.instance_of? (integer).

Tabla 13 - Métodos de Ruby

4.2.5 Ejemplo – Definición de Pruebas

En esta sección se detalla un ejemplo para ayudar a entender el lenguaje específico de dominio proporcionado por el marco de trabajo para la realización de pruebas unitarias.

El fichero “Stack.rb” contiene el código fuente correspondiente a la clase Stack (cola LIFO), la cual contiene 3 métodos. El método “push” apila un elemento al final, el método pop desapila el elemento más reciente introducido en la pila y el método “size” proporciona el número de elementos en la pila. El contenido del fichero “Stack.rb” es el siguiente:

```
# The Book of Ruby - http://www.sapphiresteel.com
# Stack - LIFO structure data
class Stack
        def initialize
                @store = Array.new
        end
        def pop
                @store.pop
        end
        def push (element)
                @store.push (element)
        end
        def size
                 @store.size
        end
end
```

Se realizarán 2 pasos de prueba, agrupados bajo un caso de prueba. Los pasos de pruebas se detallan a continuación:

- Paso de prueba 1 “Push añade elementos al final de una pila (cola LIFO): Apilar varios elementos en una pila (cola LIFO) y demostrar que el método “push” incluye elementos por el final.
- Paso de prueba 2 “Size determina el número de elementos en la cola”: Apilar varios elementos en la pila y usar el método “pop” para desapilar el último elemento de la pila y determinar cuántos elementos quedan en la pila.

Las pruebas unitarias se tienen que codificar en un fichero de texto plano o desde un editor de código fuente “.rb” (ej. Stacktests.rb). El contenido del fichero (ej. Stacktest.rb) es definido por el usuario siguiendo la sintaxis del DSL proporcionado para definición de pruebas unitarias y la sintaxis del lenguaje Ruby. El contenido del fichero “Stacktests.rb” con las pruebas unitarias podría ser el siguiente:

```
Test_Case "Stack Management" do
    The_purpose_is "to test the software defined in the file 'Stack.rb'."
    For_that "a stack is created and several items will be included."
    The "Push, Pop and Size methods provided by the Stack are used as expected."
        Test_Step "1: Push method includes the last item in a Stack." do
            Given "I have a stack with several 'a', 'b' and 'c' items." do
              @stack= Stack.new
              @stack= ["a", "b", "c"]
            end
            When "I use the Push method to include the item 'd' in the stack." do
              @stack.push ("d")
            end
            Then "'d' is the last item of the stack." do
              @last_item = @stack.pop
              @last_item.should.be.equal "d"
            end
        end
        Test_Step "2: Size method determines the number of items in the Stack." do
            Given "I have a stack with 3 items." do
              @stack = Stack.new
              @stack = ["a", "b", "c"]
            end
            When "I use the Pop method to extract an item from the stack." do
              @stack.pop
            end
            And "I use the Size method" do
             @items_in_stack = @stack.size
            end
              Then "Size methods returns 2 as the number of items in the stack." do
                @items_in_stack.should.be.equal 2
              end
        end
end
```

Es importante resaltar que el fichero debe “Stacktests.rb” debe incluir las sentencias “require_relative” para cargar el código contenido en los ficheros “Stack.rb” y “tests_tool.rb”.

4.2.6 Interfaz Usuario - Resultado de la Ejecución de Pruebas

Una vez el usuario ha definido las pruebas necesarias, siguiendo la sintaxis del DSL proporcionado y la sintaxis de Ruby, en el fichero con extensión “.rb” (ej. stacktest.rb) el usuario debe ejecutar el fichero con las pruebas unitarias (ej. C:\Sites\ruby.exe stacktests.rb). El marco de trabajo proporciona una interfaz

basada en caracteres y en la consola de comandos del sistema operativo (consola MS-DOS en nuestro caso) donde se muestra el resultado de la ejecución.

La interfaz de salida presenta cada uno de los pasos de prueba que se han ejecutado y su resultado (PASSED o FAILED), además de contabilizar el número de pasos de prueba ejecutados (test steps), el número de expectaciones (spectations), el número de fallos (failures) y el número de expectaciones erróneas (errors).

El resultado “PASSED” indica que la prueba ha sido pasada, y por lo tanto el software bajo prueba se comporta acorde a lo esperado en el paso de prueba.

El resultado “FAILED” indica que la prueba no ha sido pasada, indicando si el resultado obtenido es debido a un error o a un fallo del software. El fallo (failures <> 0) indica que el software no se comporta acorde con el comportamiento esperado en la prueba. El error (errors<> 0) indica que la expectación del paso de prueba no se ha construido acorde con la sintaxis requerida.

En la siguiente imagen (Imagen 8) se muestra un ejemplo de la interfaz de salida correspondiente a la ejecución de un paso de prueba con resultado “PASSED”. En ejemplo se puede observar que hay un paso de prueba (1 test_steps), el paso de prueba tiene una expectación (1 spectations), que no hay ningún error en la definición de las espectaciónes (0 errors), y que no ningún fallo software (0 failures).

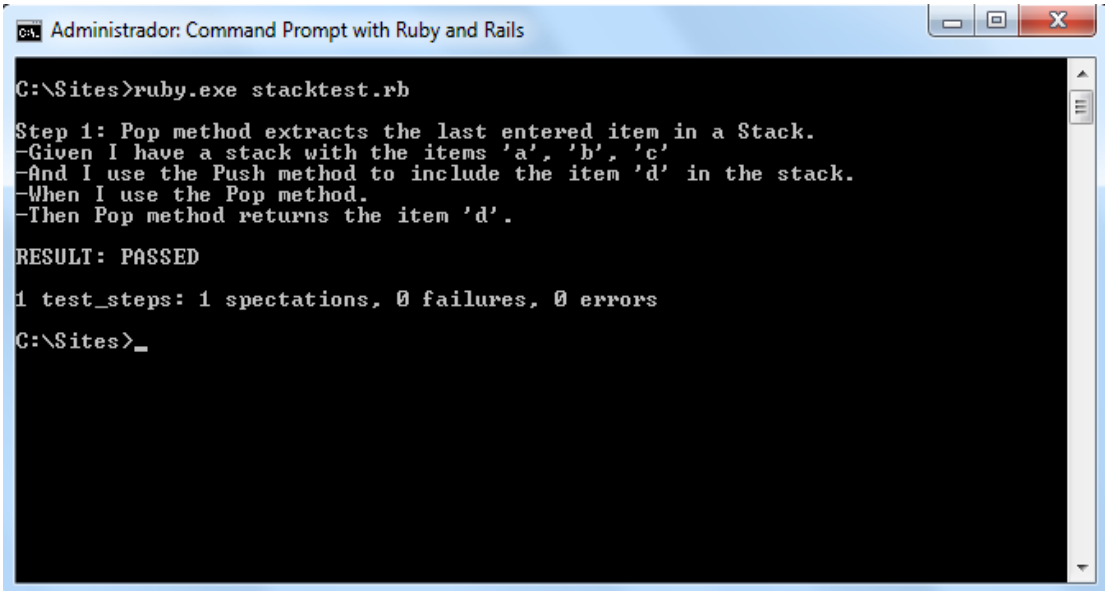


Imagen 8 – Interfaz Salida: Paso “Passed”

En la siguiente imagen (Imagen 9) se muestra un ejemplo de la interfaz de salida donde se muestran los resultados de la prueba. Dicho ejemplo corresponde a la ejecución de un paso de prueba con resultado “FAILED” porque el software no se comporta según lo esperado. En ejemplo se puede observar que hay un paso de prueba (1 test_steps), el paso de prueba tiene una expectación (1 spectations), que no hay ningún error en la definición de las expectaciones (0 errors), y que hay un fallo (1 failures), indicando que el software no se comporta según se espera en la prueba. Se presenta la etiqueta [FAILED] para diferenciar esta situación de otras situaciones de fallo.

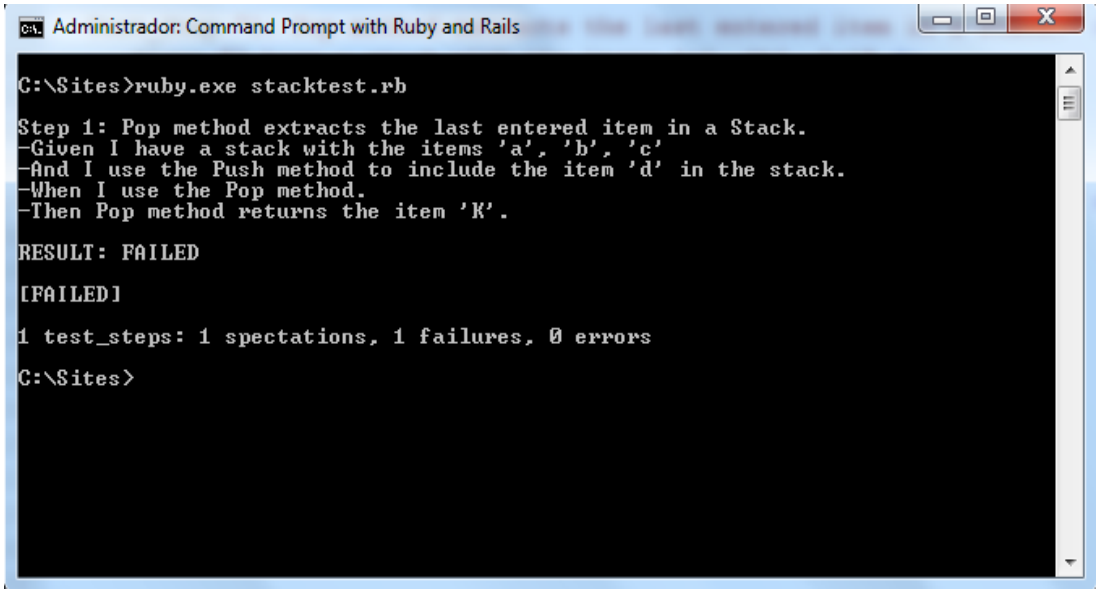


Imagen 9 – Interfaz Salida: Paso "Failed"

En la siguiente imagen (Imagen 10) se muestra un ejemplo de la interfaz de salida donde se muestran los resultados de la prueba. Dicho ejemplo corresponde a la ejecución de un paso de prueba con resultado "FAILED", porque no se ha definido ninguna expectación en el paso de prueba. En el ejemplo se puede observar que hay un paso de prueba (1 test_steps), el paso de prueba tiene una expectación (1 spectations), que no hay ningún fallo del software (0 failures) y que hay ningún error en la definición de la expectación (0 errors). Se presenta la etiqueta [MISSING_SPECTATION_ERROR] para indicar el tipo de error.

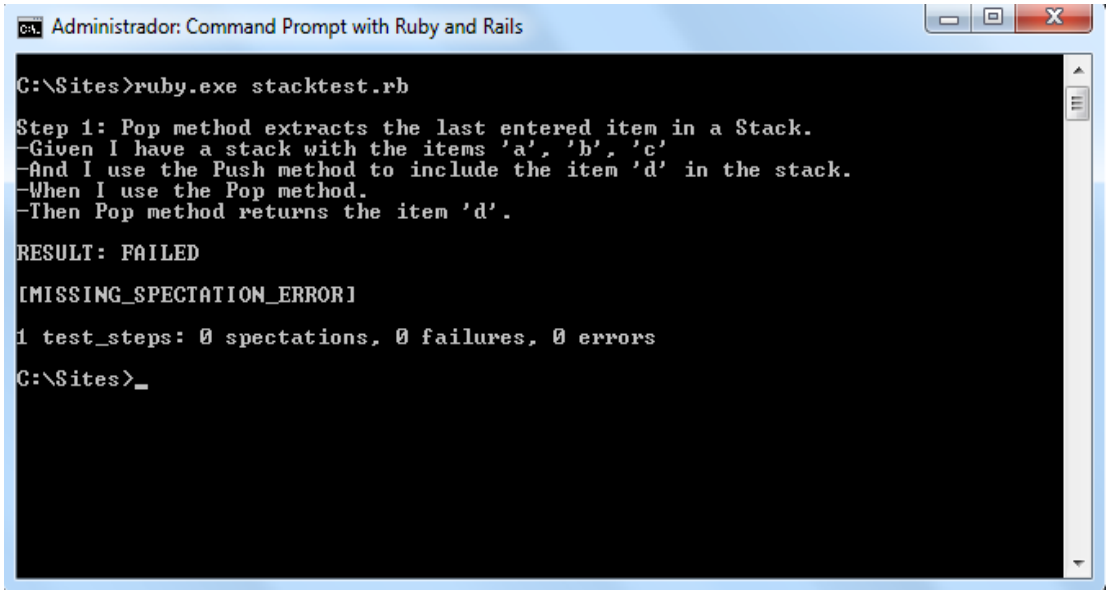


Imagen 10 – Interfaz Salida: Paso sin Expectación

En la siguiente imagen (Imagen 11) se muestra un ejemplo de la interfaz de salida donde se muestran los resultados de la prueba. Dicho ejemplo corresponde a la ejecución de un paso de prueba con resultado "FAILED" porque la prueba no ha sido correctamente definida. El motivo en cuestión es que la expectación no se ha definido acorde la sintaxis requerida. En el ejemplo se puede observar que hay un paso de prueba (1 test_steps), el paso de prueba tiene una expectación (1

spectations), que no hay ningún fallo del software (0 failures) y que hay un error en la definición de la prueba (1 errors). Se presenta la etiqueta [ERROR], junto con una descripción del tipo d error, para indicar el tipo de error.

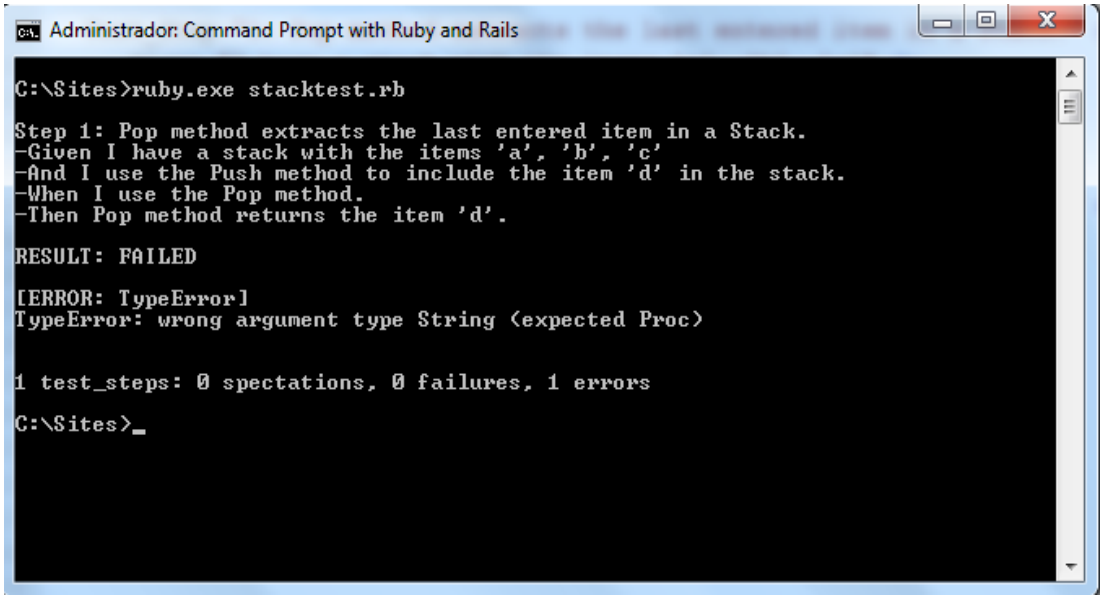


Imagen 11 – Interfaz Salida: Paso Erroneamente Definido

Los mensajes de error que se presentan corresponden con las excepciones y mensajes proporcionados por Ruby. A continuación, unos ejemplos de los errores presentados:

- TypeError: wrong argument type. El error es presentado cuando un objeto no es del tipo esperado (Ej. "texto".should.be "texto" en lugar de "texto".should.be.identical_to "texto").
- NoMethodError: undefined method. El error es presentado cuando se invoca a un método no definido (Ej. 2.should.be.equal 2 en lugar de 2.should.equal 2).
- NameError: Undefined variable. El error es presentado cuando se usa una variable no definida (E.j last_item.should.be.equal "d", siendo last_item una variable no definida).
- ArgumentError: wrong number of arguments. El error es presentado cuando los argumentos usados al invocar un método son erróneos (E.j. 4.should.be.between? 3 en lugar de 4.should.be.between? 3, 5).

La interfaz de salida también indica lo que ocurre cuando un paso de prueba no lleva el bloque (Then).

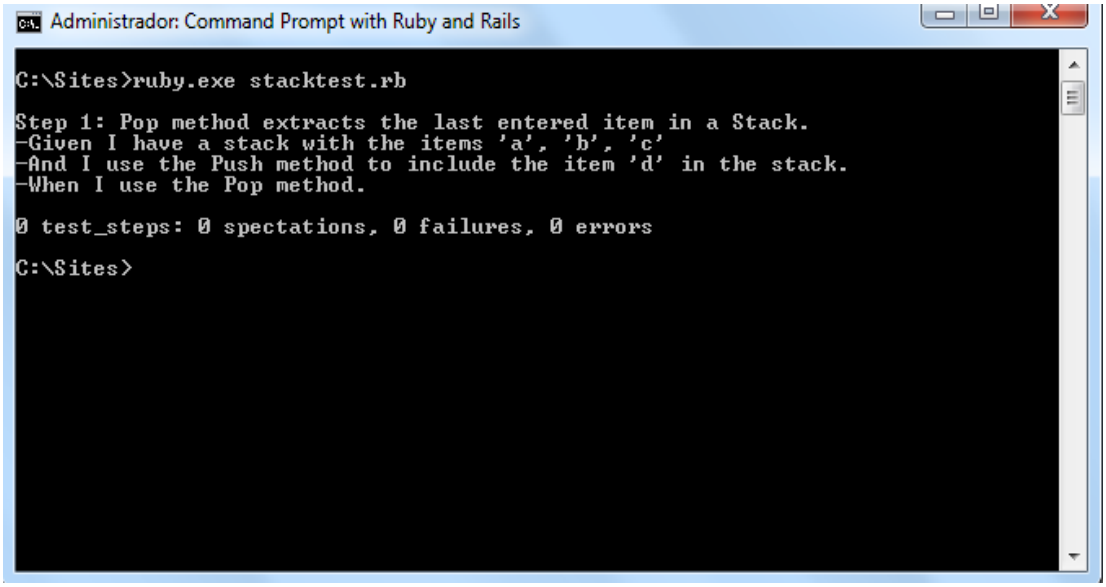


Imagen 12 – Interfaz Salda: Paso sin Bloque "Then"

4.2.7 Documentación Formal de Pruebas

Uno de los objetivos de este TFM es utilizar las pruebas unitarias definidas y ejecutadas como pruebas de aceptación y validación del software desarrollado por las partes involucradas en el proceso de desarrollo de un producto software. Para ello, además de proporciona un DSL interno basado en el lenguaje natural que permite entender de forma clara y legible las pruebas realizadas, una interfaz de salida de datos que proporciona las pruebas realizadas y el resultado de las mismas, el marco de trabajo también genera automáticamente un fichero de texto (llamado testsplan.txt) con las pruebas unitarias que se han ejecutado y el resultado de cada una de ellas (PASSED o FAILED). Este TFM propone usar el fichero "testplan.txt", que contiene las evidencias de las pruebas unitarias realizadas sobre un producto software, como mecanismo que permita la aceptación formal de un producto software por las partes involucradas.

A continuación se presenta un ejemplo del fichero testplan.txt (abierto desde MS-WORDgenerado con las pruebas realizadas en fichero "Stacktests.rb" sobre el código fuente definido en el fichero "Stack.rb", donde todas las pruebas están "PASSED" (el software se comporta acorde las pruebas definidas).

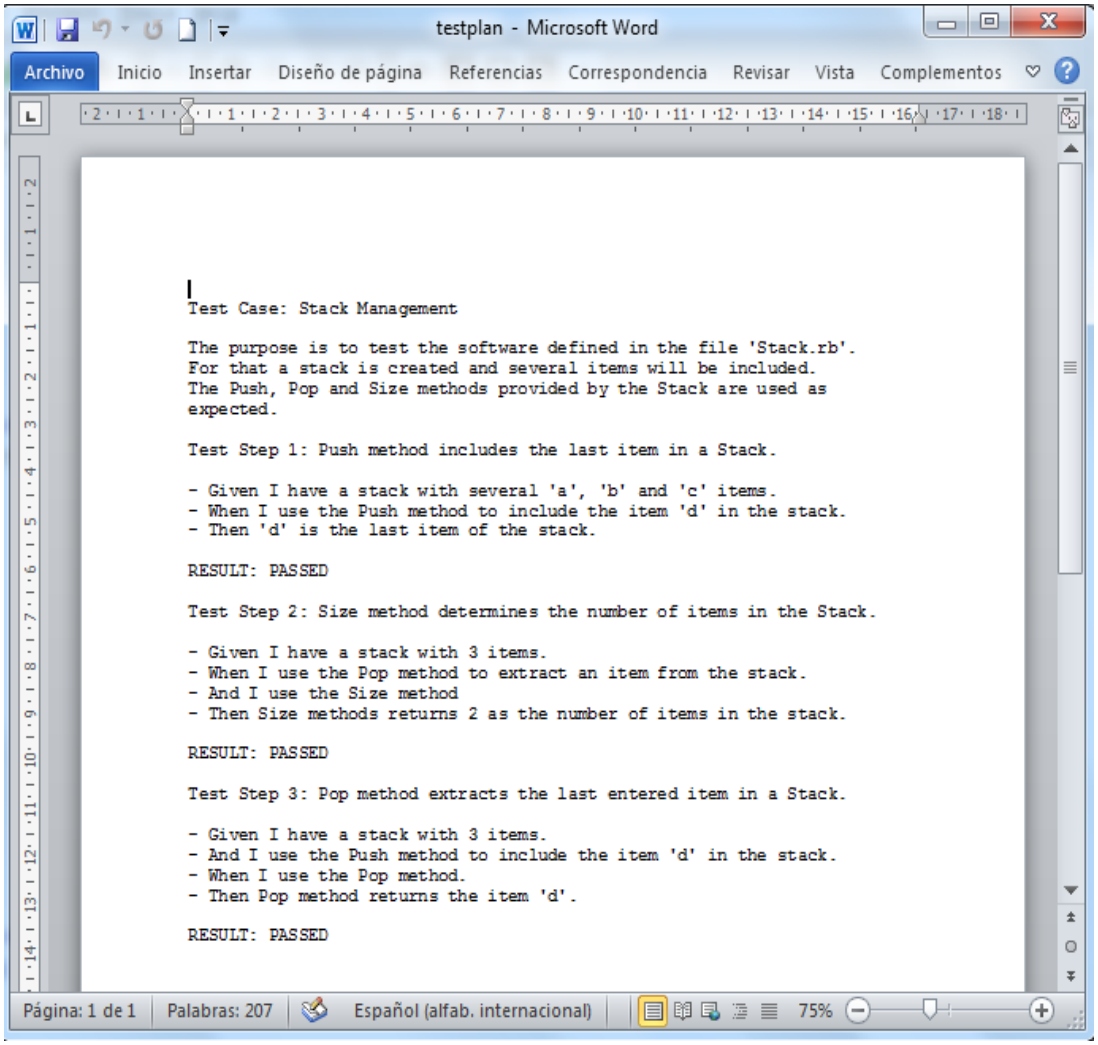


Imagen 13 – Ejemplo 1 TestPlan.txt

A continuación se presenta un ejemplo del fichero TestPlan.txt (abierto con MS-WORD) generado con las pruebas realizadas en fichero “Stacktest.rb” sobre el código fuente definido en el fichero “Stack.rb”, donde no todas las pruebas están “PASSED” (el software no se comporta acorde las pruebas definidas).

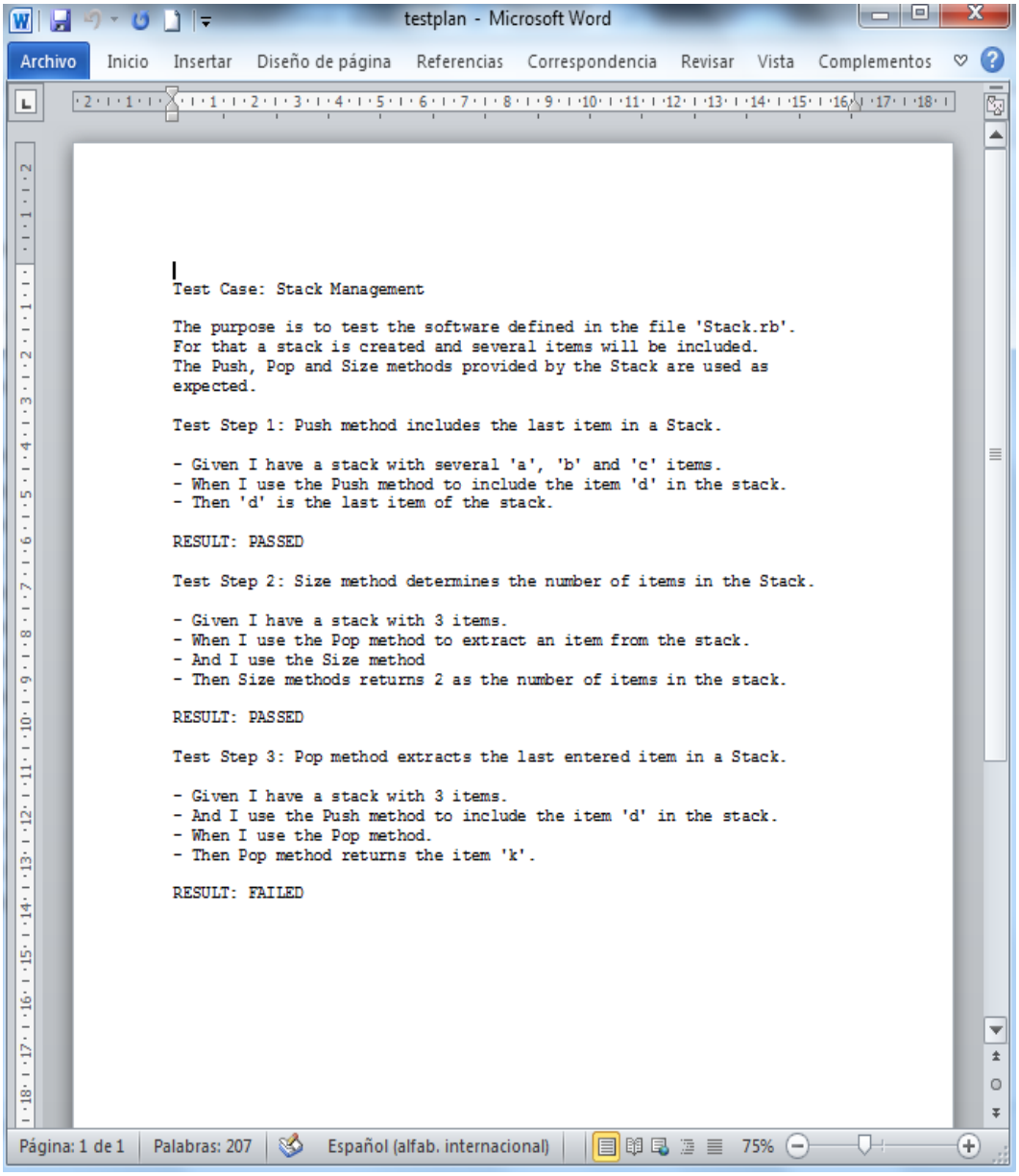


Imagen 14 – Ejemplo 2 TestPlan.txt

4.2.8 Detalles de la Herramienta “tests_tool”

La herramienta “tests_tool” es un código simple que permite la realización de pruebas unitarias en Ruby y la ejecución automática de las mismas. Además, presenta en la consola de MS-DOS el resultado de las pruebas ejecutadas y genera el fichero “testplan.txt” con las evidencias de las pruebas ejecutadas y el resultado de las mismas, estando estas descritas en lenguaje natural.

El código fuente está formado por un conjunto métodos y clases que son parte del DSL de definición de pruebas unitarias y otro conjunto de métodos y clases, que permiten la ejecución de las pruebas definidas, la captura de resultados y la generación de las evidencias con las pruebas ejecutadas.

Las clases “TestStep”, “Should”, “Error” son clases propias definidas en la aplicación y las clases Proc”, “Numeric”, “Object” son clases del kernel de Ruby cuyos métodos son extendidos en la aplicación.

La clase “TestStep” proporciona los métodos “Given”, “When”, “Then” y “And” que son parte del DSL, además de otros métodos usados internamente por la aplicación para la ejecución de los pasos de prueba y la captura de resultados (run y run_expectation):

- Los métodos “Given”, “When” and “And” ejecutan el bloque de código recibido como parámetro de entrada, y muestra la descripción recibida como parámetro de entrada en la consola MS-DOS y escriben la descripción recibida como parámetro de entrada en el fichero testplan.txt.
- El método “Then” ejecuta las expectaciones definidas dentro del bloque de código recibido como parámetro de entrada, mediante la invocación del método run_expectation.
- El método “run” ejecuta el bloque de código de un paso de prueba (invocando el método handle_test_step) y presenta en la consola MS-DOS un resumen del número de pasos de prueba ejecutadas, el número de expectaciones en los pasos de prueba, el número de errores detectados en las pruebas definidas y el número de fallos de la aplicación bajo prueba (invocando al método handle_summary). El método run es invocado desde el método Test_Step.
- El método run_expectation ejecuta las expectaciones del bloque “Then” de un paso de prueba, capturan los errores (si los hubiera) que se producen durante la ejecución de las expectaciones.

La clase Should proporciona los métodos “be”, “not”, “satisfy”, “equal”, “match”, “identical_to”, “higher_than” y “lower_than” que soportan las expectaciones proporcionadas por el DSL. Es importante resaltar, que la clase “Object” del kernel de Ruby es extendida añadiendo el método “should” público, cuya invocación crea una instancia de la clase “Should”, permitiendo el uso de los métodos expectaciones para cualquier objeto de Ruby.

La clase “Error” hereda los métodos y atributos de la clase RuntimeError, lo que permite crear errores propios cuando se produce una excepción del tipo RuntimeError durante la ejecución de las pruebas definidas.

La clase “Proc” (bloques de código tratados como variables) del Kernel de Ruby es extendida mediante la definición del método “raise” que son parte del DSL. Esto implica que cualquier instanciación de la clase Proc puede utilizar los métodos extendidos además de los propios de la clase.

La clase “Numeric” (integer, float) del Kernel de Ruby es extendida al definir el método “close” Esto implica que cualquier instanciación de la clase Numeric puede utilizar los métodos extendidos además de los propios de la clase.

La aplicación también contiene un conjunto de métodos fuera de la definición de clases, siendo alguno de ellos parte del DSL y otros se usan para dar soporte a la aplicación:

- Los métodos “Tests_Suite” y “Test_Case” ejecutan el bloque de código del caso o suite de prueba. Son parte del DSL y sirven para organizar las pruebas dentro del DSL.
- El método Test_Step lanza la ejecución del bloque de código del paso de prueba. Para ello, instancia un objeto de la clase TestStep y ejecuta el método run con el bloque de código del paso de prueba. El método

Test_Step es parte del DSL y define el contenido de la prueba unitaria a ejecutar.

- Los métodos “The_purpose_is”, “The_objetive_is”, “As_a”, “I_wanto_to”, “In_order_to”, “It_is_important”, “In_addition”, “For_that”, “The”, “Even” y “Although” son parte del DSL y se usan para describir la narrativa de los casos y suite de pruebas dentro del DSL. El código de estos métodos es muy simple, dirigido a la escritura de la descripción del método en el fichero de “testplan.txt”
- El método handle_summary se utiliza para presentación del resumen del número de pasos de prueba ejecutados, el número de expectaciones en los pasos de prueba, el número de errores detectados en las pruebas definidas y el número de fallos de la aplicación bajo prueba.
- El método handle_test_step se utiliza para ejecutar el paso de prueba, es decir, el ejecutar el bloque de código Given/When/Then).
- El método handle_expectaciones se utiliza de ejecutar la expectación de cada paso de prueba y presentar el resultado de la ejecución.

Es importante resaltar que el código fuente del marco de trabajo Bacon (analizado en este TFM) ha sido usado como referencia para la implementación de la herramienta Tests_Tool tal que parte del código fuente de Bacon [1] relacionado con la gestión de las expectaciones es reutilizado en la herramienta Tests_Tool.

Para más detalle consultar el fichero “test_tool.rb” suministrado junto a esta memoria, donde se describen con más detalle los métodos y clases definidos en el código fuente.

4.2.9 Especificaciones de Uso

Como requisitos previos al uso de la aplicación Tests_Tool, se requiere la instalación del compilador Ruby 2.3.3 para sistema operativo Windows. Se recomienda usar RubyInstaller (versión 2.3.6 and 2.3.3. released) para la instalación del compilador. RubyInstaller puede descargarse en https://rubyinstaller.org/.

Una vez instalado el compilador, se requiere la instalación de las librerías de Ruby:

- Rubygems. Descargar "Rubygems" del siguiente enlace: http://rubygems.org/gems/rubygems y proceder a su instalación.

Para la instalación de las citadas librerías ejecutar los siguientes comandos desde la consola de comandos MS-DOS y desde el directorio donde el ejecutable Ruby.exe este localizado:

- C:/Sites> gem install rubygems

Una vez instalado Ruby, hay que proceder a la copia del fichero tests_tool.rb proporcionado en este TFM en el directorio C:/Sites.

Una vez el usuario ha definido las pruebas unitarias (ej. Stacktests.rb) sobre el código fuente a probar (ej. Stack.rb), el usuario del marco del trabajo puede proceder a la ejecución de las mimas. Para la ejecución de las pruebas unitarias, hay que invocar al ejecutable Ruby.exe junto con el nombre del fichero con las pruebas unitarias definidas.

Ejemplo: C:/Sites>Ruby.exe Stacktests.rb.

Como resultado de la ejecución del fichero con las pruebas unitarias definidas por el usuario, el marco de trabajo presentará en la consola de comandos de MS-DOS el resultado de la ejecución de las pruebas unitarias y generará el fichero TestPlan.txt

(con las evidencias de las pruebas ejecutadas) en el mismo directorio donde se copió la herramienta Tests_Tool.rb.

4.3 Casos de Prueba Aplicados para las Pruebas de Demostración

En la sección se detallan los casos de prueba definidos para la demostración del uso del prototipo presentado en esta investigación.

4.3.1 PRUEBA 1

El fichero Stack.rb contiene el siguiente código fuente Ruby:

```
# The Book of Ruby - http://www.sapphiresteel.com
# Stack - LIFO structure data
class Stack
        def initialize
                @store = Array.new
        end
        def pop
                @store.pop
        end
        def push (element)
                @store.push (element)
        end
        def size
                 @store.size
        end
 end
```

Se realizarán 3 pasos de prueba, agrupados bajo un caso de prueba. Los pasos de pruebas se detallan a continuación:

- Paso de prueba 1 “Push añade elementos al final de una pila (cola LIFO): Apilar varios elementos en una pila (cola LIFO) y demostrar que el método “push” incluye elementos por el final.
- Paso de prueba 2 “Size determina el número de elementos en la cola”: Apilar varios elementos en la pila y usar el método “pop” para desapilar el último elemento de la pila y determinar cuántos elementos quedan en la pila.
- Paso de prueba 3 “Pop extrae el último elemento introducido en la pila”: Apilar varios elementos en la pila y demostrar que el método “pop” extrae de la pila el último elemento introducido en la pila.

Las pruebas unitarias estarán definidas en el fichero “Stacktests.rb”, con el siguiente contenido:

```
require_relative 'Stack.rb'
require_relative 'tests_tool.rb'


Test_Case "Stack Management" do
```

```
The_purpose_is "to test the software defined in the file 'Stack.rb'."
For_that "a stack is created and several items will be included."
The "Push, Pop and Size methods provided by the Stack are used as expected."
    Test_Step "1: Push method includes the last item in a Stack." do
        Given "I have a stack with several 'a', 'b' and 'c' items." do
            @stack= Stack.new
            @stack= ["a", "b", "c"]
        end
        When "I use the Push method to include the item 'd' in the stack." do
            @stack.push ("d")
        end
        Then "'d' is the last item of the stack." do
            @last_item = @stack.pop
            @last_item.should.be.equal "d"
        end
    end
    Test_Step "2: Size method determines the number of items in the Stack." do
        Given "I have a stack with 3 items." do
            @stack = Stack.new
            @stack = ["a", "b", "c"]
        end
        When "I use the Pop method to extract an item from the stack." do
            @stack.pop
        end
        And "I use the Size method" do
          @items_in_stack = @stack.size
        end
        Then "Size methods returns 2 as the number of items in the stack." do
            @items_in_stack.should.be.equal 2
        end
    end
    Test_Step "3: Pop method extracts the last entered item in a Stack." do
        Given "I have a stack with 3 items." do
            @stack = Stack.new
            @stack = ["a", "b", "c"]
        end
        And "I use the Push method to include the item 'd' in the stack." do
            @stack.push "d"
        end
        When "I use the Pop method." do
```

```
            @last_entered_item = @stack.pop
          end
          Then "Pop method returns the item 'd'." do
            @last_entered_item.should.be.equal "d"
          end
        end
end
```

El resultado de ejecutar el fichero StackTests.rb (ej. C:/Sites>Ruby.exe Stacktests.rb) desde la ventana de comandos de MS-DOS, proporciona la siguiente salida de datos en la consola MS-DOS.

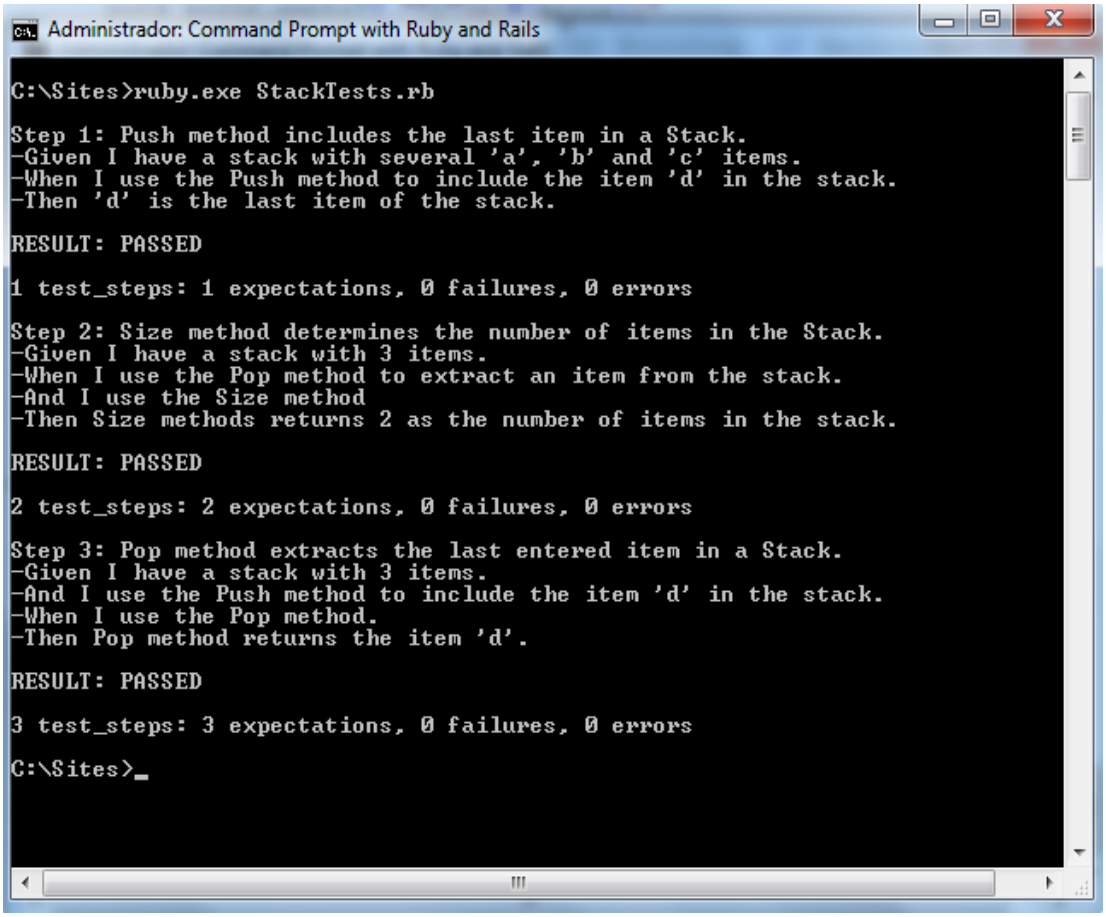


Imagen 15 – Prueba 1: Salida Consola

Además, la herramienta "Test_Tool.rb" genera el fichero testplan.txt, en el mismo directorio donde está copiado la herramienta Test_Tool.tb, con el siguiente contenido:

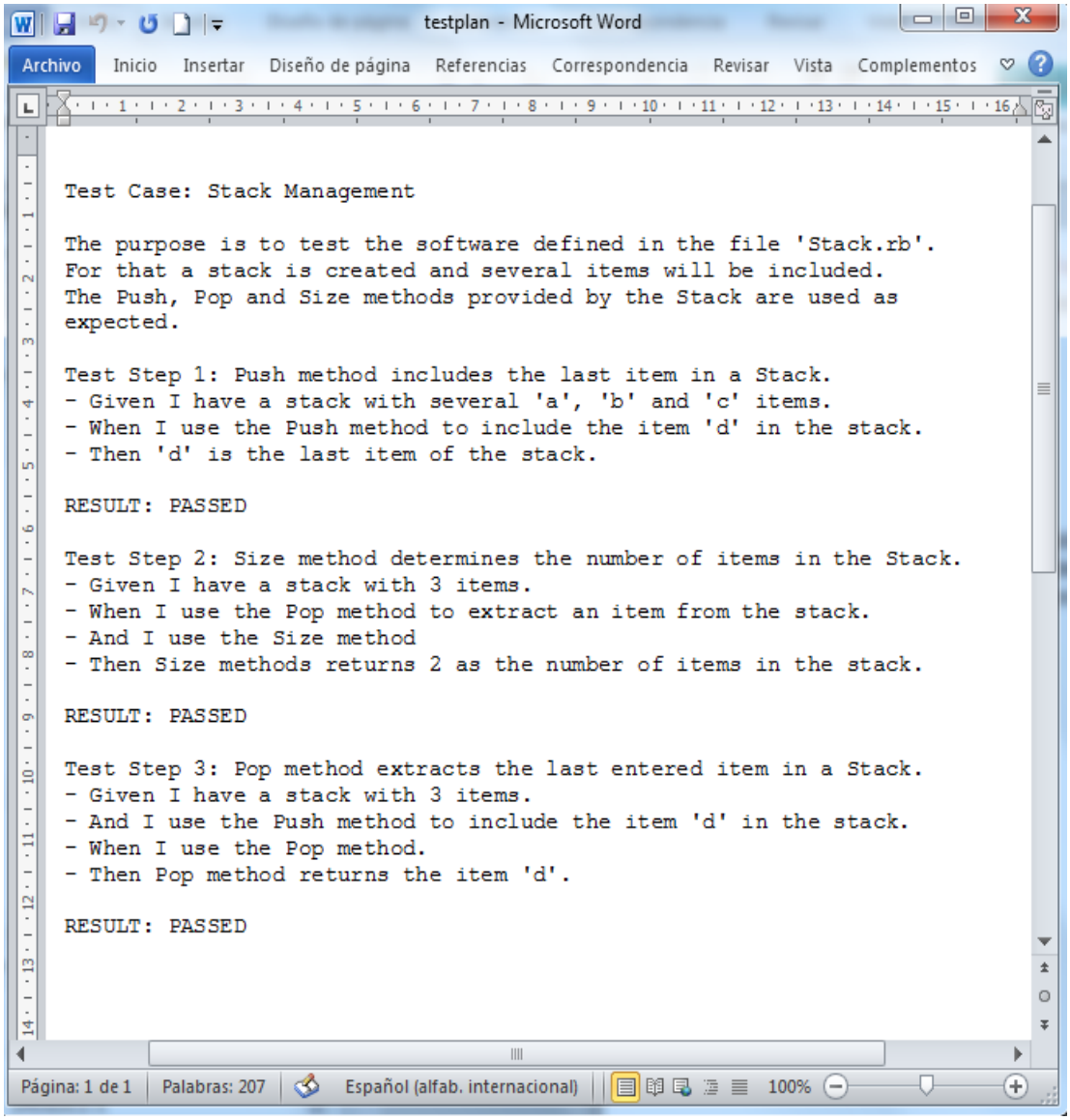


Imagen 16 – Prueba 1: TestPlan.txt

4.3.2 PRUEBA 2

El fichero "Operaciones.rb" contiene el siguiente código fuente Ruby:

```
#!/usr/bin/env ruby
class SimpleNumber
    def add (x, y)
            x + y
    end
    def multiply (x, y)
            x * y
    end
    def subt (x, y)
            x - y
    end
```

```
        end
```

Se realizarán 2 casos de prueba, agrupados bajo una de prueba Suite de Pruebas, llamada “Operaciones Matemáticas”. Cada caso de prueba está formado por 2 pasos de prueba:

- Caso 1: “Operaciones Simples”, con los siguientes pasos de prueba
  - Paso de Prueba 1: 3 + 5 = 8
  - Paso de Prueba 2: 3*5!= 10
  - Paso de Prueba 3: 200 – 33 = 167
- Caso 2: “Operaciones Complejas”, con los siguientes pasos de prueba
  - Paso de Prueba 4: (3+5) * 8 < 65
  - Paso de Prueba 5: (5 +2) * (3*2) > 40

Las pruebas unitarias estarán definidas en el fichero “OperacionesTests.rb”, con el siguiente contenido:

```
require_relative 'operaciones.rb'
require_relative 'tests_tool.rb'

Tests_Suite "Mathematical Operations" do
  I_want_to "to test the software defined in the file 'Operations.rb'."
  For_that "several tests cases are created to test simple and complex operations."
  Test_Case "Simple Operations" do
    The_purpose_is "to test simple operations."
    For_that "Add, Multiply and Subt methods are used individually."
        Test_Step "1. Add: 3 + 5 = 8." do
            Given "I have 3." do
              $a = 3
            end
            And "I have 5 more." do
              $b = 5
            end
            When "I add them." do
              @result = SimpleNumber.new.add ($a,$b)
            end
            Then "the result should be equal 8." do
              @result.should.be.equal 8
            end
        end
        Test_Step "2. Multiply: 3 * 4 = !10." do
            Given "I have 3." do
              $a = 3
            end
```

```
            When "I multiply it by 4 " do
              @result = SimpleNumber.new.multiply ($a,$b)
            end
            Then "the result should not be equal 10." do
              @result.should.not.be.equal 10
            end
        end
        Test_Step "3. Subt: 200- 33 = 167." do
            Given "I have 200." do
              $a = 200
            end
            When "I subtract 33 from 200." do
              @result = SimpleNumber.new.subt ($a,33)
            end
            Then "the result should be 167." do
              @result.should.be.equal 167
            end
        end
    end
Test_Case "Complex Operations" do
    The_purpose_is "to test complex operations."
    For_that "Add, Multiply and Subt methods are used to solve complex operations."
        Test_Step "4. Operation: (3 + 5)*8 < 65." do
            Given "I add 3 and 5" do
              $f =  SimpleNumber.new.add (3,5)
            end
            When "I multiply it by 8." do
              @result = SimpleNumber.new.multiply ($f,8)
            end
            Then "the result should lower than 65." do
              @result.should.be.lower_than 65
            end
        end
        Test_Step "5. Operacion: (5+2)*(3*2) > 40" do
            Given "I add 5 and 2 to obtain 7" do
              $g = SimpleNumber.new.add (5,2)
            end
            And "I multiply 3 and 2 to obtain 6." do
              $h = SimpleNumber.new.multiply (3,2)
            end
```

```
            When "I multiply the 7 and 6." do
              @result =  SimpleNumber.new.multiply ($g,$h)
            end
            Then "the result should be higher than 40." do
              @result.should.be.higher_than 40
            end
        end
    end
end
```

El resultado de ejecutar el fichero OperacionesTests.rb (ej. C:/Sites>Ruby.exe OperacionesTests.rb) desde la ventana de comandos de MS-DOS, proporciona la siguiente salida de datos en la consola MS-DOS.

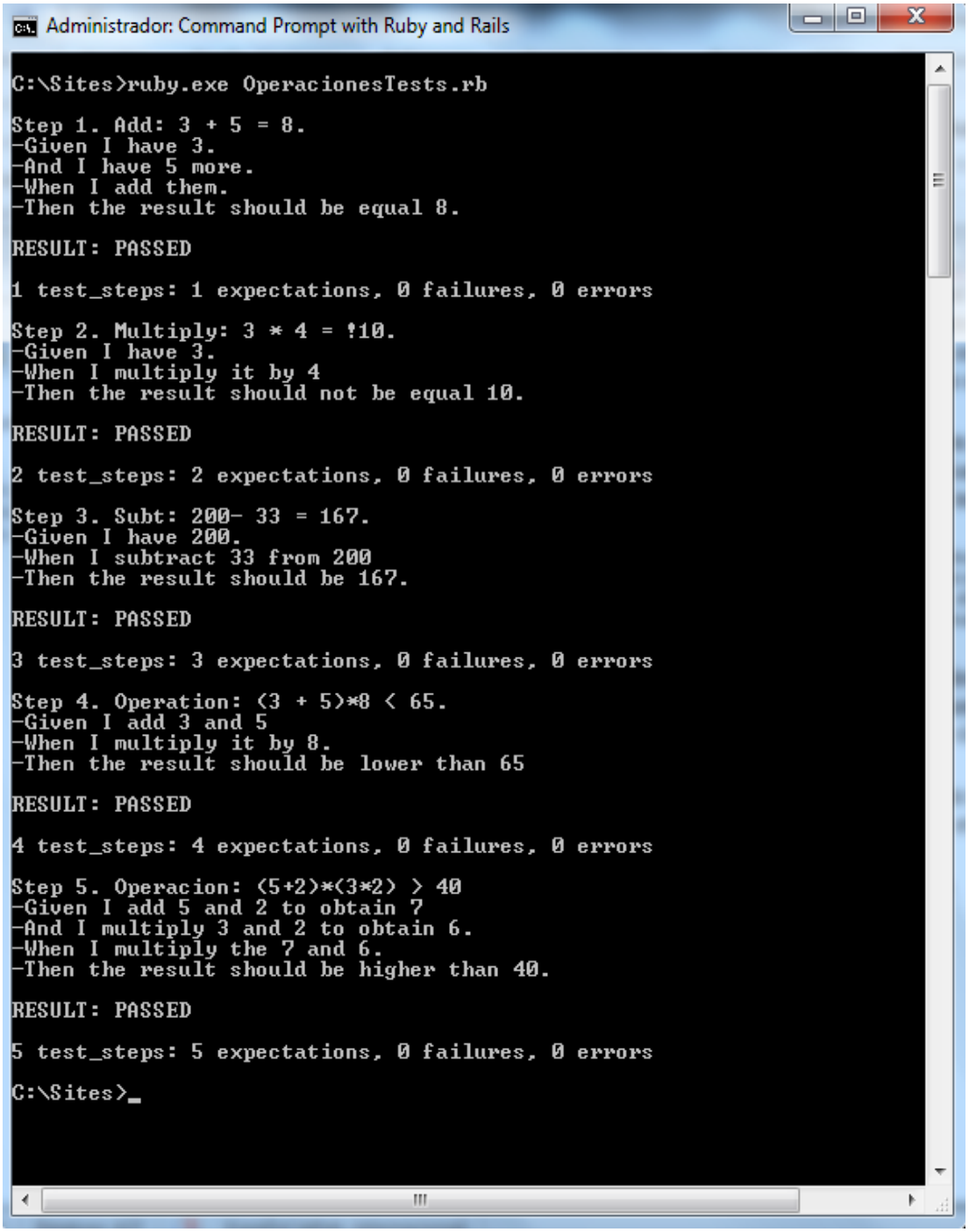


Imagen 17 – Prueba 2: Salida Consola

Además, la herramienta “Test_Tool.rb” genera el fichero testplan.txt, en el mismo directorio donde está copiado la herramienta Test_Tool.tb, con el siguiente contenido:

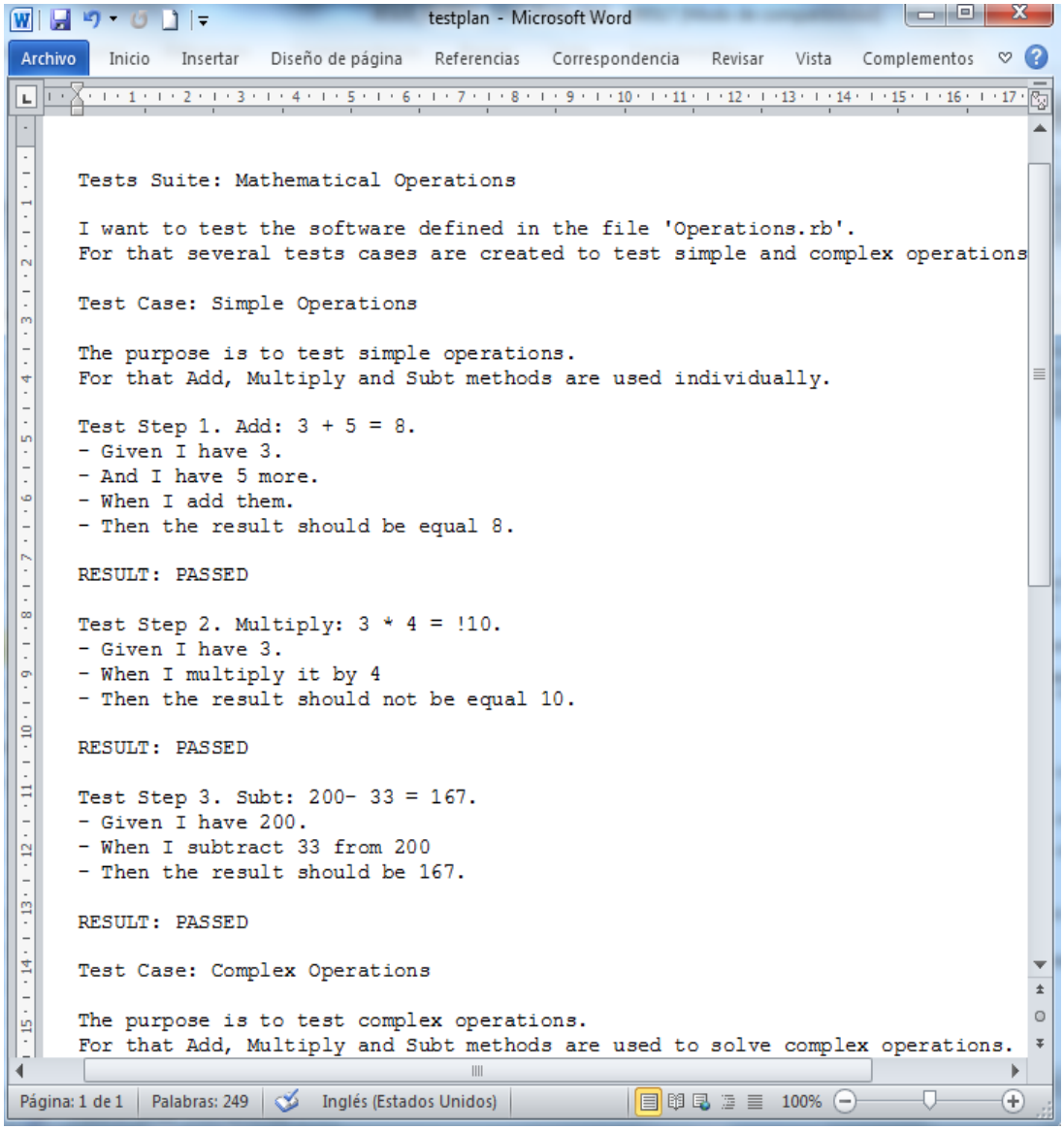


Imagen 18 – Prueba 2: TestPlan.txt (1)

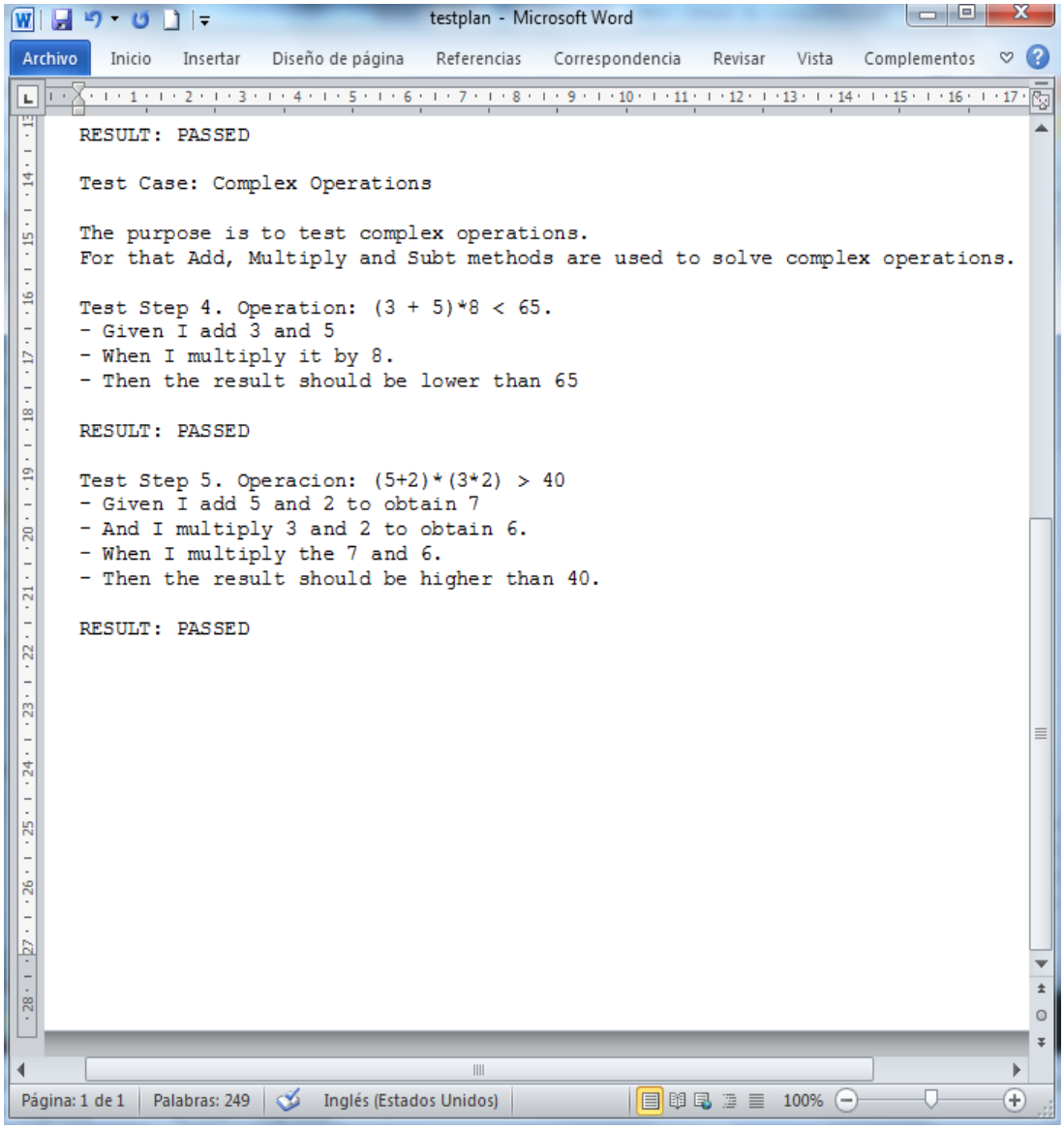


Imagen 19 – Prueba 2: TestPlan.txt (2)

4.3.3 PRUEBA 3

El fichero Operaciones.rb contiene el siguiente código fuente Ruby:

```ruby
#!/usr/bin/env ruby
class SimpleNumber
    def add (x, y)
        x + y
    end
    def multiply (x, y)
        x * y
    end
    def subt (x, y)
        x - y
```

```
        end
        def div (x, y)
                x / y
        end
    end
```

El fichero Stack.rb contiene el siguiente código fuente Ruby:

```
# The Book of Ruby - http://www.sapphiresteel.com
 # Stack - LIFO structure data
 class Stack
        def initialize
                @store = Array.new
        end
        def pop
                @store.pop
        end
        def push (element)
                @store.push (element)
        end
        def size
                 @store.size
        end
 end
```

Se realizará una Suite de Pruebas con dos casos de prueba. Un caso de prueba contendrá los pasos de prueba para probar el código fuente del fichero “Stack.rb” y otro caso de prueba para probar el código fuente del fichero “Operaciones.rb”.

El caso de prueba para probar el código fuente del fichero “Stack.rb” contendrá los siguientes pasos pruebas:

- Paso de prueba 1: “Push añade elementos al final de una pila (cola LIFO): Apilar varios cadenas de texto en una pila (cola LIFO) y demostrar que el método “push” incluye elementos por el final. Para ello, se apilarán las cadenas de texto “leon” y “gato”, mediante el método Push se apilará la cadena de texto “tigreton”, y se determinará que el último elemento apilado se ajusta al patrón “tigre”.
- Paso de prueba 2: “Size determina el número de elementos en la cola”: Apilar varios elementos en la pila y usar el método “pop” para desapilar el último elemento de la pila y determinar cuántos elementos quedan en la pila.
- Paso de prueba 3: “Pop extrae el último elemento introducido en la pila”: Apilar varios elementos en la pila y demostrar que el método “pop” extrae de la pila el último elemento introducido en la pila.

El caso de prueba para probar el código fuente del fichero “Operaciones.rb” contendrá los siguientes pasos pruebas:

- Paso de Prueba 4: 20/5 = 4.
- Paso de Prueba 5: 10/0 genera la excepción ZeroDivisionError.
- Paso de Prueba 6: El resultado de (200 – 33) está en un delta de 3 respecto a 170.

Las pruebas unitarias estarán definidas en el fichero “StackOperacionesTests.rb”, con el siguiente contenido:

```
require_relative 'operaciones.rb'
require_relative 'stack.rb'
require_relative 'tests_tool.rb'

Tests_Suite "Stack.rb and Operations.rb files" do
   As_a "tester."
   I_want_to "validate the Stack.rb and Operaciones.rb files."
   In_order_to "make a delivery of the created software."

   Test_Case "Stack Management" do
   The_purpose_is "to test the software defined in the file 'Stack.rb'."
   For_that "a stack is created and several items will be included."
   The "Push, Pop and Size methods provided by the Stack are used as expected."
         Test_Step "1: Push method includes the last item in a Stack." do
               Given "I have a stack with the strings 'gato' and 'leon'." do
                     @stack = Stack.new
                     @stack = ["gato", "leon"]
               end
               When "I use the Push method to include the string 'tigreton' in the stack." do
                     @stack.push ("tigreton")
               end
               Then "the last item of the stack should match 'tigre'." do
                     @stack.pop.should.match (/tigre/)
               end
         end
         Test_Step "2: Size method determines the number of items in the Stack." do
               Given "I have a stack with 3 items." do
                  @stack = Stack.new
                  @stack = ["a", "b", "c"]
               end
               When "I use the Pop method to extract an item from the stack." do
                  @stack.pop
               end
               And "I use the Size method." do
```

```
            @items_in_stack = @stack.size
          end
          Then "Size methods returns 2 as the number of items in the stack." do
            should.satisfy {@items_in_stack == 2}
          end
      end
      Test_Step "3: Pop method extracts the last entered item in a Stack." do
          Given "I have a stack with 3 items" do
            @stack = Stack.new
            @stack = ["a", "b", "c"]
          end
          And "I enter the item 'c' in the stack by using the Push method." do
            @stack.push ("d")
          end
          When "I use the Pop method." do
            @last_entered_item = @stack.pop
          end
          Then "Pop method returns the item 'd'." do
            should.satisfy {@last_entered_item == "d"}
          end
      end
end
Test_Case "Mathematical Operations" do
I_want_to "to test the software defined in the file 'Operations.rb'."
For_that "several tests cases are created to test simple and complex operations."
      Test_Step "4. div: 20 / 5 = 4." do
          Given "I have 20." do
            $a = 20
          end
          When "I divide it by 5." do
            @result = SimpleNumber.new.div ($a,5)
          end
          Then "the result should be identical to 4." do
            @result.should.be.identical_to 4
          end
      end
      Test_Step "5. Div: 10/0 raises ZeroDivisionError Exception." do
          Given "I have 10 items" do
            $a = 10
          end
```

```
            When "I divide it by 0." do
              $e = lambda {SimpleNumber.new.div ($a,0)}
            end
            Then "the system shoud raise ZeroDivisionError Exception." do
                $e.should.raise (ZeroDivisionError)
            end
        end
        Test_Step "6. Subt: 200 - 33 is close to 170 inside a delta of 3." do
            Given "I have 200." do
              $a = 200
            end
            When "I subtract 33 from 200." do
              @result = SimpleNumber.new.subt ($a,33)
            end
            Then "the result should be close to 170 inside a delta of 3." do
              @result.should.be.close (170,3)
            end
        end
    end
end
```

El resultado de ejecutar el fichero StackOperacionesTests.rb (ej. C:/Sites>Ruby.exe StackOperacionesTests.rb) desde la ventana de comandos de MS-DOS, proporciona la siguiente salida de datos en la consola MS-DOS.

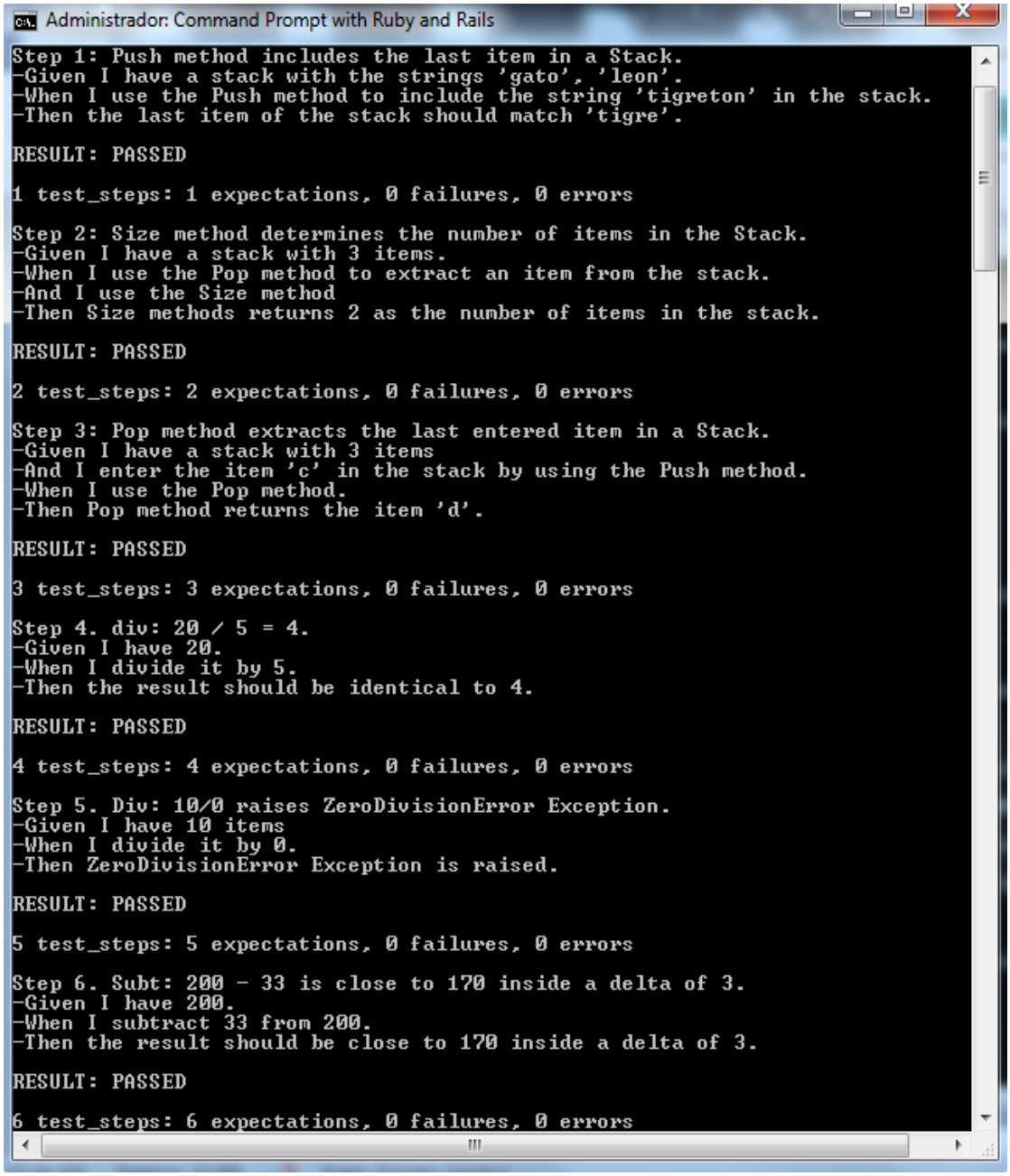


Imagen 20 – Prueba 3: Salida Consola

Además, la herramienta "Test_Tool.rb" genera el fichero testplan.txt, en el mismo directorio donde está copiado la herramienta Test_Tool.tb, con el siguiente contenido:

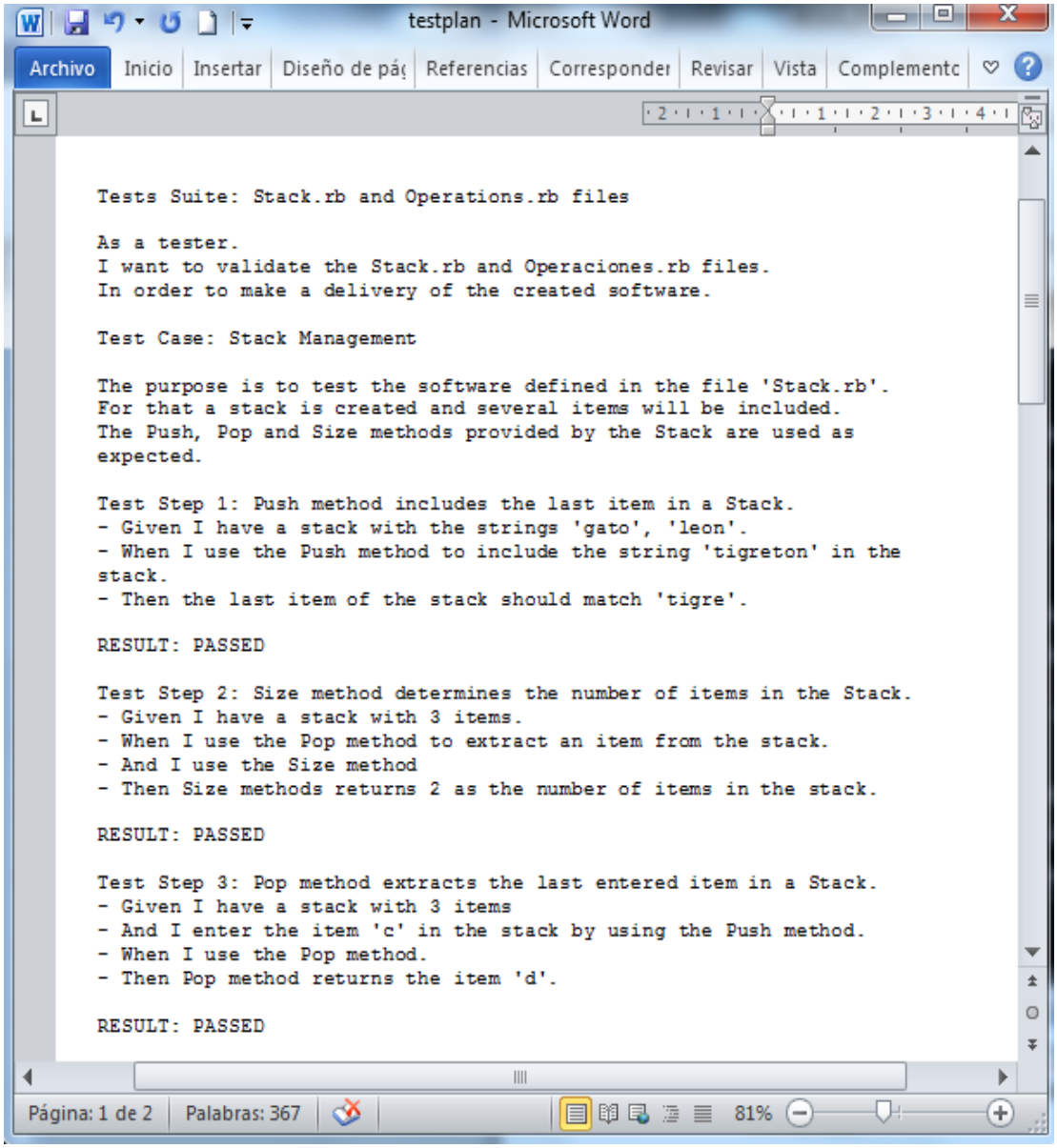


Imagen 21 – Prueba 3: TestPlan.txt (1)

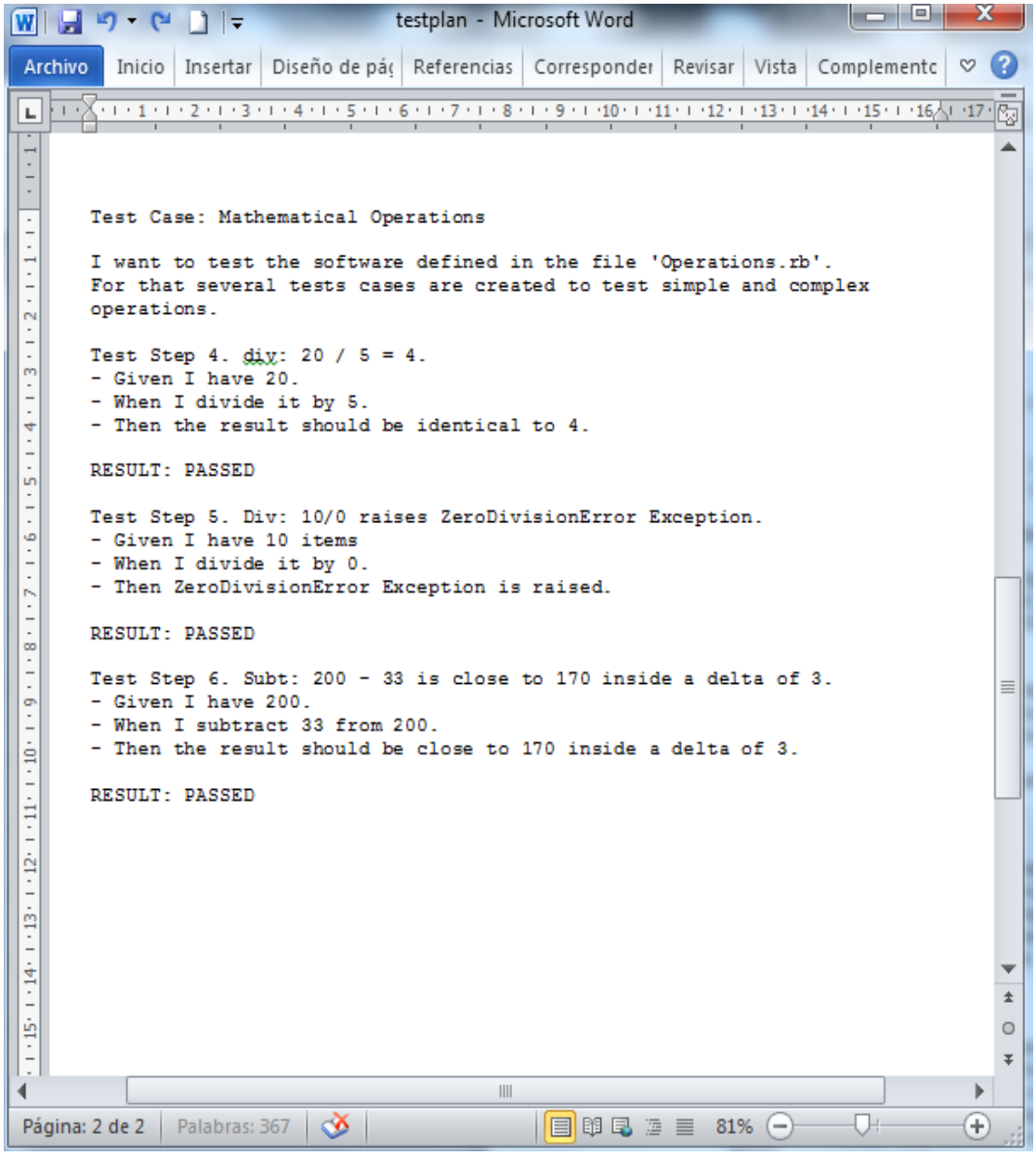


Imagen 22 – Prueba 3: TestPlan.txt (2)

5. EVALUACIÓN Y COMPARACIÓN DE LA SOLUCIÓN

Una vez realizado el proceso de investigación de los marcos de trabajo de pruebas unitarias frente al problema planteado en esta investigación, habiendo ya sentado las bases para la evaluación de las soluciones y conociendo al detalle la solución propuesta por esta investigación, es momento de proceder a la evaluación y comparación de la solución propuesta y las soluciones ya existentes en el mercado.

El procedimiento para la evaluación y comparación se basa en dar respuesta a las siguientes preguntas:

1. ¿El proceso de definición de las pruebas y el proceso de desarrollo del código fuente para la automatización de dichas pruebas son procesos separados?, es decir, ¿el usuario define las pruebas con una herramienta y las codifica mediante otra herramienta?
2. ¿Las pruebas definidas son legibles y comprensibles por el personal técnico, el no técnico y por el cliente?
3. ¿Permite la codificación de pruebas usando un lenguaje propio de pruebas y cercano al lenguaje natural?
4. ¿La solución proporciona un mecanismo de documentación formal de las pruebas realizadas y que puedan ser utilizadas para la aceptación del software desarrollado?

En la siguiente tabla se muestra el resultado de la evaluación y comparación de las distintas soluciones respecto a las preguntas plantadas.

Evaluación / Comparativa	¿El proceso de definición de las pruebas y el proceso de desarrollo del código fuente para la automatización de dichas pruebas son procesos separados?	¿Las pruebas definidas son legibles y comprensibles tanto por personal técnico como por personal no técnico?	¿Permite la codificación de pruebas usando un lenguaje propio de pruebas y cercano al lenguaje natural?	¿La solución proporciona un mecanismo de documentación formal de las pruebas realizadas y que puedan ser utilizadas para la aceptación del software desarrollado?
Test/Unit	NO	NO	NO	NO
Rspec	NO	NO	NO	NO
Minitest/Unit	NO	NO	NO	NO
Minitest/Spec	NO	NO	NO	NO
Bacon	NO	NO	NO	NO
Wrong	NO	NO	NO	NO
Riot	NO	NO	NO	NO
Cucumber	SI	SI	NO	SI
Should/Context	NO	NO	NO	NO
Coulda	NO	NO	SI	NO
Shindo	NO	NO	NO	NO
Tests_Tool (*)	NO	SI	SI	SI

(*) Tests_Tool es la herramienta desarrollada en este TFM.

Tabla 12 - Comparativa y Evaluación de los Marcos de Trabajos y DSLs analizados

A continuación se detalla la justificación de cada una de las respuestas mostradas en la tabla anterior:

1. ¿El proceso de definición de las pruebas y el proceso de desarrollo del código fuente para la automatización de dichas pruebas son procesos separados?, es decir, ¿el usuario define las pruebas con una herramienta y las codifica mediante otra herramienta?

Todos los marcos de trabajo analizados son herramientas muy potentes y útiles para la automatización de pruebas unitarias. El público al que va dirigido dichas herramientas son los desarrolladores de aplicaciones con lenguaje RUBY.

En Test/Unit, RSpec, Minitest/Unit, Minitest/Spec Bacon, Wrong, Riot, Shindo, Shoulda/Context y Coulda no hay separación entre el proceso de definición de pruebas y el proceso de programación del código fuente para la automatización de las pruebas unitarias. El desarrollador define las pruebas y codifica las pruebas en un mismo proceso y en un mismo fichero “rb”. El interfaz proporcionado por las soluciones es un editor de texto plano editor de código fuente donde el usuario tiene que definir y programar las pruebas unitarias que quiere realizar.

En Cucumber, la definición de pruebas y la programación del código fuente para la automatización de las pruebas son procesos separados que se realizan desde diferentes utilidades. La definición de las pruebas se realiza mediante un DSL externo, a través de los ficheros de características (ficheros de texto plano interpretados por Cucumber) y la programación del código para la automatización de las pruebas se realiza a través de los ficheros de pasos (ficheros de texto plano con extensión “rb”).

La separación entre los procesos de definición de pruebas y de programación del código fuente para la automatización de las pruebas ralentiza la realización de pruebas unitarias al tener que realizar dos procesos de manera separada en lugar de solo uno, además elevar la complejidad para la realización de las mismas ya que es necesario conocer y gestionar dos procesos diferenciados.

La solución propuesta en este TFM no separa los procesos de definición de las pruebas del proceso de programación del código fuente para la automatización de las pruebas. La definición de pruebas y codificación de las mismas se realizan desde un mismo fichero “rb.”

2. ¿Las pruebas definidas son legibles y comprensibles tanto por personal técnico como por personal no técnico?

En el proceso de desarrollo de un producto software es muy importante la detección temprana de errores, tanto errores funcionales como errores de entendimiento del producto requerido. Por ello es importante, que las pruebas unitarias realizadas sean fácilmente comprensibles y legibles por las partes involucradas en el desarrollo software (ej. analistas de negocio, programadores, probadores, personal técnico o no técnico, etc.).

Las pruebas definidas mediante Test/Unit, RSpec, Minitest/Unit, Minitest/Spec, Bacon, Wrong, Riot, Shindo, Shoulda/Context, y Coulda) no son legibles ni entendibles por personal no técnico. La definición/programación de pruebas unitarias con dichas herramientas no deja de ser programación en lenguaje Ruby. Es cierto, que el grado de conocimiento del lenguaje Ruby para comprender las pruebas realizadas varía de unos a otros dependiendo del DSL proporcionado para

codificar pruebas más o menos cercanos del lenguaje natural. En un extremo estaría Test-Unit, donde es necesario tener unos conocimientos más profundos de lenguaje Ruby para comprender las pruebas y en el otro extremo estaría Coulda, donde se requiere un conocimiento menor debido al DSL proporcionado para la codificación de las pruebas.

Cucumber dispone de los ficheros de características (features) donde el usuario define las pruebas unitarias usando un lenguaje Gherkin y el lenguaje natural para la definición de los escenarios de prueba. Es importante destacar, que los ficheros de definición de pasos, no son compresibles por personal no técnico ya que en ellos se desarrolla el código necesario para la ejecución de los escenarios definidos en los ficheros de características.

La solución propuesta en este TFM genera el fichero “testplan.txt” con las pruebas que se han ejecutado, descritas en lenguaje natural. Por lo tanto, las pruebas son legibles y comprensibles tanto por personal técnico como no técnico.

3. ¿Permite la codificación de pruebas usando un lenguaje propio de pruebas y cercano al lenguaje natural?

El uso de un lenguaje especifico lo más cercano natural para la definición del lenguaje natural facilita el entendimiento de las pruebas realizadas, además de facilitar el proceso de codificación ya que se utiliza un lenguaje cercano al que usamos las personas para comunicarnos. Los marcos de trabajo y DSL analizados proporcionan lenguajes específicos más o menos cercanos al lenguaje natural.

Test/Unit, Minitest/Unit, Riot, Wrong y Shindo no proporcionan lenguaje cercano al lenguaje natural. La codificación de pruebas consiste en usar las utilidades del lenguaje Ruby y en la invocación de los métodos (Asertos en la mayoría de los casos) que permiten valorar el comportamiento del software bajo prueba.

Shoulda/Context es un DSL para trabajar con Test/Unit y Minitest/Unit que permite un cierto acercamiento a la definición de pruebas basadas en asertos al lenguaje natural. En mi opinión, requiere mejoras para acercarse todavía más la definición de pruebas unitarias al lenguaje natural.

RSpec, Minitest/Spec, Bacon y Shindo permiten la codificación de pruebas unitarias utilizando un lenguaje específico más cercano al lenguaje natural que otros marcos de trabajo, debido al uso de expectaciones (ej. must, should, expect, etc.) y uso de métodos que definir el contexto de una prueba. Según sus creadores, dichos marcos trabajos permiten describir el comportamiento del software de una manera muy cercana al lenguaje natural, como si estuviésemos hablando al cliente o a otro desarrollador. No obstante, en mi opinión requieren mejoras para alcanzar dicha meta.

La codificación de pruebas en Cucumber consiste en usar el lenguaje Ruby para definir el contenido de los métodos definidos por el usuario desde el fichero de características. No proporciona un mecanismo similar a asertos y expectaciones para determinar si el comportamiento del software bajo prueba está acorde al comportamiento esperado.

Coulda permite la codificación de pruebas unitarias con un lenguaje cercano al lenguaje natural. No obstante, el uso de métodos asertos en lugar de métodos expectaciones para la valorar el comportamiento del software bajo prueba no permite al usuario claramente asociar la prueba codificada con el resultado esperado. El uso de expectaciones en lugar de asertos acercaría más a Coulda al ideal de usar el lenguaje natural para la definición de pruebas.

Ej: Uso de expectación “must.be” en lugar del “assert”:

```
Scenario "Size Method” do
  Given "I have a stack." do
    @stack = Stack.new
  end
  When "I use the Push method to include the items 'a' and 'b' " do
    @stack.push (“a”)
    @stack.push (“b”)
  end
  Then "the stack size must be '2' " do
    assert (2, @stack.size) # Mejora con Expectaciones: @stack.size.must.be 2
  end
```

La agrupación y organización de pruebas es importante cuando el número de pruebas a realizar es grande. No todos los DSL analizados permiten una agrupación de pruebas con diferente categorización (ej. casos de prueba y suite de pruebas). De los marcos de trabajo analizados solamente Test/Unit y Minitest permite la agrupación de pasos pruebas en distintas categorías mediante la definición de casos de prueba y suite de pruebas. El resto permite de agrupación de pasos de prueba, pero usando la misma categorización o etiqueta (ej. Anidamiento de bloques “describe” o “context”).

La solución propuesta en TFM permite la codificación de pruebas unitarias con un lenguaje cercano al lenguaje natural. La definición de los pasos de prueba a través de los métodos “Given”, “And”, “When”, “Then” y el uso de métodos expectaciones basadas en el método “Should” para valorar el comportamiento del software permite la codificación de pruebas unitarias con un lenguaje muy cercano al lenguaje natural. La definición de los métodos Tests_Suite, Test_Case y Test_Step permite la definición de pruebas unitarias utilizando un lenguaje propio de los entornos de pruebas, además de permitir la agrupación de pruebas.

4. ¿La solución proporciona un mecanismo de documentación formal de las pruebas realizadas que pueda ser utilizado para la aceptación del software desarrollado?

En el proceso de aceptación de un producto software es muy importante tener evidencias sobre las pruebas realizadas y que estas describan de manera legible y compresible el correcto comportamiento del producto desarrollado.

Test/Unit, RSpec, Minitest, Bacon, Wrong, Riot, Shindo, Shoulda, Coulda), excepto en Cucumber RSpec, MiniTest y Test-Unit no proporcionan un mecanismo que sirva para formalizar con las todas las partes involucradas las pruebas realizadas y que puedan servir para la aceptar el código desarrollado. Las pruebas automatizadas mediantes estas soluciones no dejan de ser código fuente desarrollado en RUBY, que no es comprensible por las partes involucradas.

Cucumber proporciona un mecanismo que permite presentar las pruebas realizadas de manera entendible por todas las partes involucrados en el desarrollo (desarrolladores, probadores, cliente, etc.) para formalizar las pruebas realizadas y pueden ser usado para la aceptación del código desarrollado. Los escenarios que se definen con Cucumber representan claramente y de manera entendible las pruebas realizadas sobre software desarrollado.

La solución propuesta en TFM genera el documento “testplan.txt” con las pruebas realizadas (descritas en lenguaje natural) y el resultado de la ejecución de las

mismas. Dicho documento constituye un mecanismo para formalizar las pruebas realizadas, pudiendo ser usado para la aceptación del código desarrollado.

6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1 Conclusiones

El origen de esta investigación surge tras empezar a conocer el mundo de la automatización de las pruebas unitarias de Ruby y detectar la posibilidad de realizar pruebas unitarias automáticas que sean legibles y comprensibles por el personal involucrado, bien sea personal técnico o no. Además, de usar las pruebas realizadas, para facilitar la aceptación del software desarrollado por las partes involucradas.

Por ello, surge la necesidad de investigar los marcos de trabajo de pruebas de Ruby más populares con el fin buscar alguna herramienta que cubriera mis expectativas. En base a la investigación realizada, se puede concluir que los marcos de trabajo de pruebas unitarias para Ruby analizados son herramientas muy potentes para la automatización de pruebas unitarias, dirigidas específicamente a desarrolladores de software. Dichas herramientas son utilizadas para probar el software ya desarrollado o para el desarrollo de software dirigido por pruebas o por comportamiento. El desarrollo de pruebas unitarias con estas herramientas consiste en programar código ejecutable usando los métodos y funciones que proporcionan dichas herramientas y el lenguaje de programación Ruby para la realización de las pruebas, desde de un editor de texto o editor de código. Algunas de estas herramientas (caso de RSpec, Minitest/Spec, Coulda y Cucumber) proporcionan un lenguaje de dominio específico (DSL) que facilita la definición de las pruebas y la legibilidad y la comprensión de las pruebas realizadas por personal técnico (desarrolladores principalmente), facilitando el mantenimiento y comprensión del software desarrollado.

De las herramientas analizadas, solamente Cucumber, permite entender claramente, tanto al personal técnico y no técnico, las pruebas realizadas sobre un software desarrollado gracias al lenguaje de dominio proporcionado para la definición de prueba. Por ello, las pruebas realizadas con Cucumber pueden ser usadas para la aceptación del software por las partes involucradas. Cucumber presenta el inconveniente de separar el proceso de definición de escenarios de pruebas (usados como documentación formal de pruebas) y del proceso de programación del código fuente para la automatización de las pruebas, lo que implica que el usuario deba definir las pruebas y codificar las pruebas en procesados separados.

Las aportaciones realizadas en esta investigación se centran en el desarrollo de un prototipo para la automatización de pruebas unitarias que permite la definición de pruebas utilizando un lenguaje específico cercano al lenguaje natural lo que facilita la realización de pruebas unitarias. Además, también permite la definición de pruebas legibles y entendibles por las partes involucradas (tanto personal técnico como no técnico) en el desarrollo de un proceso software y facilitar al usuario la definición y codificación de las pruebas mismas en un solo proceso. Para ello proporciona un lenguaje de dominio específico interno basado en el lenguaje natural y en la jerga usada en los entornos de prueba y un fichero de texto con las pruebas realizadas expresadas en lenguaje natural.

También permite usar las pruebas definidas como un mecanismo de aceptación del código por las partes involucradas gracias a la legibilidad y claridad de las pruebas definidas mediante el prototipo.

6.2 Aplicación de Conocimientos Adquirido en el Master

Los conceptos esenciales en los que se basa este trabajo de fin de master, son los siguientes:

- Lenguaje de Programación Ruby.
- Pruebas Unitarias en Ruby
- Frameworks de Pruebas Unitarias
- DSLs
- Metaprogramación en Ruby (programas que escriben programas).

Las asignaturas “Desarrollo de Líneas de Productos Software mediante un Enfoque Generativo” y de “Generación automática de Código” me han permitido iniciarme en el mundo de la programación en Ruby, siendo el lenguaje usado para la implementación de la solución.

Dichas asignaturas también me ha permitido adquirir conocimientos de lenguajes de dominio, siendo la asignatura la “Desarrollo de Líneas de Productos Software mediante un Enfoque Generativo” la que más ha aportado a dichos conocimientos. Dicha asignatura me ha ensañado las cualidades deseables de un DSL, la sintáxis y semántica de un DSL, tipos de DSLs (externos o embebidos en lenguajes de propósito general) y los elementos necesarios para su implementación (ej. analizadores, generadores, etc.).

La asignatura “Desarrollo de Líneas de Productos Software mediante un Enfoque Generativo” ha sido el punto de partida para adquirir conocimientos de metaprogramación en Ruby.

La asignatura “Generación Automática de Código” me ha permitido descubrir el mundo de las pruebas unitarias y la generación automática de código automática, lo que ha dado lugar a la realización de este trabajo fin de master.

Los conocimientos aquiridos en las asignaturas de Arquitecturas orientadas a servicios, Gestión y Mejora de Procesos Software y Especificación de los sistemas software no han sido de aplicación directa en el desarrollo de este TFM. No obstante, los conomientos adquiridos en dichas asignaturas me han servido para ser más crítico en el analisis de los marcos de trabajo de pruebas unitarias de Ruby y dar forma a este TFM.

6.3 Trabajos Futuros

Los trabajos futuros son de gran diversidad, dado el carácter de prototipo de la herramienta Tests_Tool. El prototipo presentado es la base para un posterior desarrollo de una herramienta que permita realizar pruebas unitarias bajo los conceptos desarrollados en el prototipo.

Estas tareas futuras consisten en completar la herramienta en aquellos aspectos que no están desarrollados y en aquellos que no están totalmente desarrollados, entre los que destacan los siguientes aspectos.

- Proporcionar APIs para realizar pruebas de rendimiento (benchmarking).
- Completar la herramienta para que genere un fichero “.docx” con las pruebas realizadas con un formato específico más propio de documentos formales de pruebas.
- Proporcionar más expectaciones para facilitar la realización de pruebas sobre cualquier tipo de software.

7. REFERENCIAS Y BIBLIOGRAFÍA

1. Pruebas unitarias

   http://es.wikipedia.org/wiki/Prueba_unitaria

   http://en.wikipedia.org/wiki/Unit_testing

   http://artofunittesting.com/definition-of-a-unit-test/

   http://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728

   https://www.ruby-toolbox.com/categories/testing_frameworks

2. Tutorial TDD, BDD y Test de Aceptación.

   http://www.adictosaltrabajo.com/tutoriales/tdd-bdd-test-de-aceptacion/

3. Test-Unit.

   http://en.wikibooks.org/wiki/Ruby_Programming/Unit_testing

   https://github.com/test-unit/test-unit

   http://rubytutorial.wikidot.com/unit-testing

   http://www.rubydoc.info/gems/test-unit/3.0.9/Test/Unit/Assertions

   http://ruby-doc.org/stdlib-1.8.7/libdoc/test/unit/rdoc/Test/Unit.html

4. RSpec

   http://rspec.info/

   https://github.com/rspec/rspec

   https://www.tutorialspoint.com/rspec/

   http://tutorials.jumpstartlab.com/topics/internal_testing/rspec_and_bdd.htm

   http://blog.teamtreehouse.com/an-introduction-to-rspec

   ChelimmsKy, Daniel; North, Dam. The RSpec Book. Behavior-Driven Development with RSpec, Cucumber and Friends, 2010. http://zepho.com/rails/books/the-rspec-book.pdf

5. Cucumber.

   http://en.wikipedia.org/wiki/Cucumber_(software)

   https://github.com/cucumber/cucumber/wiki/Tutorials-and-Related-Blog-Posts

ChelimmsKy, Daniel; North, Dam. The RSpec Book. Behavior-Driven Development with RSpec, Cucumber and Friends, 2010. http://zepho.com/rails/books/the-rspec-book.pdf

6. Minitest.

   https://github.com/seattlerb/minitest

   http://ruby-doc.org/stdlib-2.0.0/libdoc/minitest/rdoc/MiniTest.html

   http://www.rubyinside.com/a-minitestspec-tutorial-elegant-spec-style-testing-that-comes-with-ruby-5354.html

   http://blog.teamtreehouse.com/short-introduction-minitest

   http://www.mattsears.com/articles/2011/12/10/minitest-quick-reference

7. Bacon.

   https://github.com/chneukirchen/bacon

   http://www.rubydoc.info/gems/bacon/1.2.0/

   [1] https://github.com/chneukirchen/bacon/blob/master/lib/bacon.rb

8. Wrong.

   https://github.com/sconover/wrong

9. Riot

   https://github.com/thumblemonks/riot

   https://rubygems.org/gems/riot/versions/0.12.7

10. Shoulda-Context

    https://github.com/thoughtbot/shoulda-context

    http://www.rubydoc.info/github/thoughtbot/shoulda-context/master/Shoulda/Context/ClassMethods

11. Shindo

    https://github.com/geemus/shindo

12. Coulda

    https://rubygems.org/gems/coulda/versions/0.7.1

    https://github.com/elight/coulda

13. Historia de Usuario

https://www.ebgconsulting.com/blog/using-given-when-then-to-discover-and-validate-requirements-2/

https://es.wikipedia.org/wiki/Historias_de_usuario

http://jmbeas.es/guias/historias-de-usuario/

http://www.angellozano.com/requisitos-del-sistema-vs-casos-uso-vs-historias-usuario/

14. Programación y Metaprogración en Ruby

Perrota, Paolo. Metaprograming Ruby, 2010.

Cooper, Peter. Beginning Ruby “From Notice to Professional”, 2009.

Collingbourne, Huw. The Book of Ruby, 2009.

8. GLOSARIO DE TÉRMINOS

A

API – Application Programming Interface es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.……13

B

BDD –Behaviour-Driven Development o desarrollo guiado por comportamiento es un proceso de desarrollo de software que surgió a partir del desarrollo guiado por pruebas (DGP). El desarrollo guiado por el comportamiento combina las técnicas generales y los principios del DGP, junto con ideas del diseño guiado por el dominio y el análisis y diseño orientado a objetos para proveer al desarrollo de software y a los equipos de administración, con herramientas compartidas y un proceso compartido de colaboración en el desarrollo de software……1 4. 13, 22, 30, 32, 38, 49, 59, 105, 108.

E

EBNF– Extended Backus –Naur Form. es un metalenguaje usada para expresar gramáticas libres de contexto. ……13, 108

M

MS-DOS –Microsoft Disk Operating System……74, 79, 80, 81, 84, 88, 95, 108
MS-WORD ·– Microsoft Word……78, 108

T

TDD –Test-Driven Development o desarrollo guiado por pruebas es una práctica de ingenieria del software para el desarrollo de software que consiste escribir primero las pruebas y se verifica que las pruebas fallan. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido. ……13, 105, 108
TFM –Trabajo Fin de Master……14, 15, 16, 55, 60, 77, 81, 100, 101, 102, 108

X

XML –· Extensible Markup Language or "Lenguaje de Marcas Extensible", es un meta-lenguaje que permite definir lenguajes de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos en forma legible……14, 108