Ingeniería de Software

12 - Testing

IIC2143-3

Josefa España

jpespana@uc.cl

Testing

Proceso usado para evaluar la **correctitud, completitud** y **calidad** de un programa.

- puede incluir actividades con el fin de encontrar errores en el programa, con el fin de ser corregidos antes de lanzar el producto a los usuarios.

¿Por qué es importante?

Los bugs en software pueden ser costosos e incluso peligrosos:

¿Qué causa los defectos?

Las personas cometen errores, somos falibles:

- Errores de programadores: errores en el diseño, construcción, falta de criterios, etc.
- Errores de usuarios: usan el software de forma no esperada.

Clasificaciones

- **Error**: acción humana que produce un resultado incorrecto.
- **Defecto**: presencia de una imperfección que puede ocasionar fallas.
- **Falla**: comportamiento observable incorrecto con respecto a los requisitos.

Un **error** induce un **defecto** en el software, que se presenta a través de una **falla** en las pruebas.

¿Qué puede salir mal?

Varios factores pueden llegar a ser defectos o fallas:

- Errores en la especificación, diseño, e implementación.
- Errores en el uso del sistema.
- Condiciones del medio ambiente.
- Daño intencional.
- ...

¿Por qué testing?

- Todos cometemos y cometerémos errores:
  - Detectar los defectos de forma temprana facilita el desarrollo.
  - Reparar los defectos es más barato si son encontrados antes.

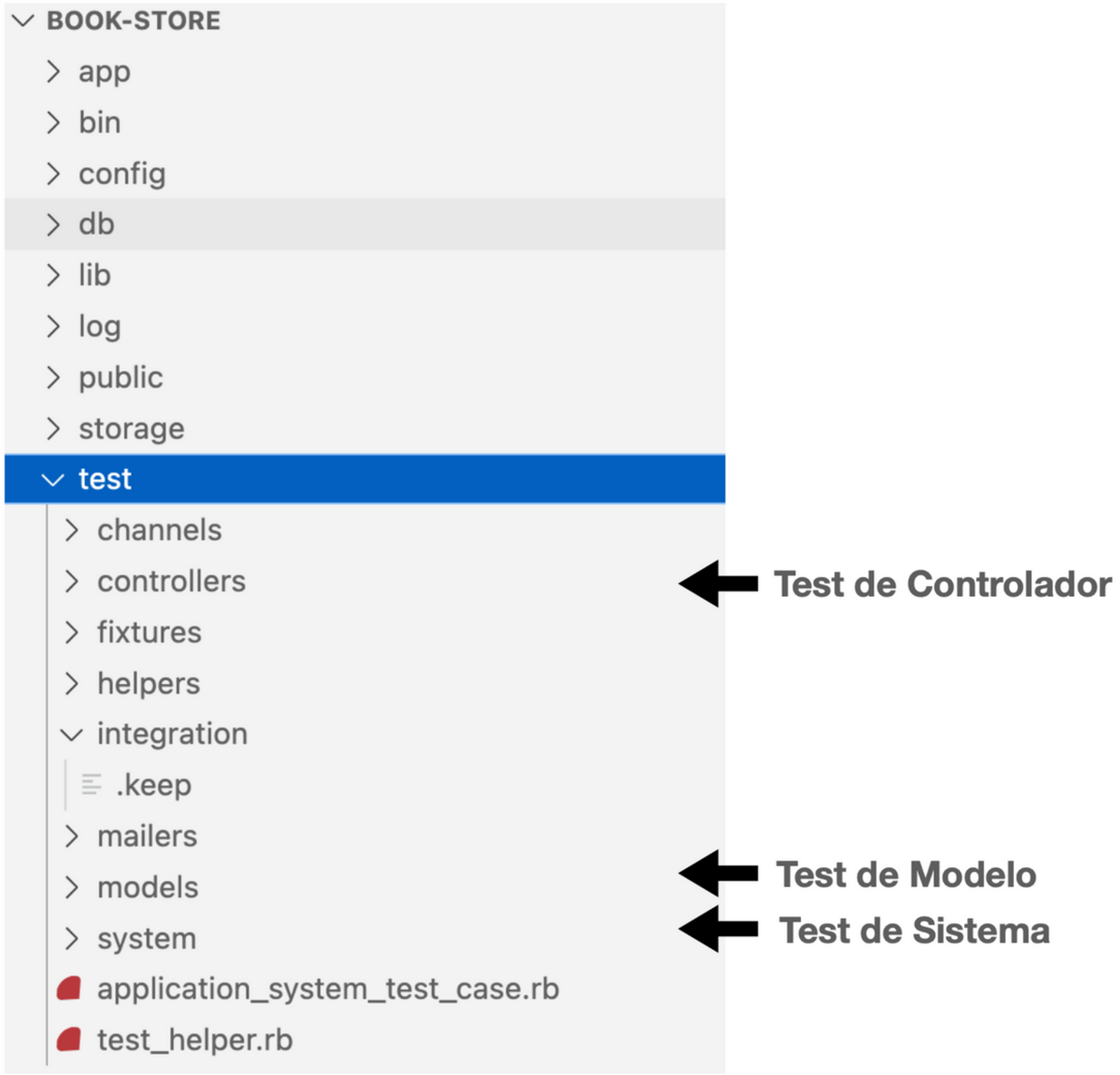
- Si algo puede fallar, fallará.

- Queremos que el equipo de trabajo revise que todo sigue funcionando al agregar un nuevo comportamiento, de manera rápida y confiable.

Testing con Rails

Testing en diferentes niveles

- **Unit testing**: En Rails, trata de verificar si las clases del modelo que creamos funcionan correctamente en isolación (por si solas, sin interacción de otras).
- **Integration testing:** En Rails, trata de verificar si los controladores creados en el proyecto están funcionando de manera correcta.
- **System testing**: En Rails, evalúa la funcionalidad del sistema como un todo. Es como abrir la página web, e interactuar con ella para ver que todo funcione. Se puede hacer manual o automática.


BOOK-STORE
app
bin
config
db
lib
log
public
storage
test
channels
controllers
fixtures
helpers
integration
.keep
mailers
models
system
application_system_test_case.rb
test_helper.rb
Test de Controlador
Test de Modelo
Test de Sistema

Unit test - Ejemplo

```ruby
class Book < ApplicationRecord
    validates :title, presence: true
    validates :author, presence: true
    validates :year, presence: true
end
```

Modelo

```ruby
class BookTest < ActiveSupport::TestCase
  test "should not save without title" do
    @book = Book.new(title:"",author:"Juan P.", year:2023)
    result = @book.save
    assert_not result,"saved the title without title"
  end
end
```

Test

Unit test - Estructura

```ruby
class BookTest < ActiveSupport::TestCase
  test "should not save without title" do
    # inicialización, donde crean los objetos y datos necesarios
    @book = Book.new(title:"",author:"Juan P.", year:2023)

    # estimulo, donde se realiza la acción a evaluar
    result = @book.save

    # verificación, donde se evalúa si el resultado es el esperado
    assert_not result,"saved the book without title"
  end
end
```

Unit test - Ejecución

```
>rails test test/models/book_test.rb
Running 1 tests in a single process (parallelization
threshold is 50)
Run options: --seed 53778
# Running:
.
Finished in 0.008640s, 115.7407 runs/s, 115.7407
assertions/s.
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

Consideraciones

- Rails tiene 3 bases de datos: una para pruebas, otra de producción, y otra para desarrollo. (esto se puede ver en config/database.yml).

- Rails inicializará una base de datos dedicada para ejecutar cada prueba, de esta forma cada test se ejecuta de forma separada y aislada.

Tipos de asserts

- assert(test, [msg])
- assert_not(test, [msg])
- assert_equal(expected, actual, [msg])
- assert_not_equal(expected, actual, [msg])
- assert_same(expected, actual, [msg])
- assert_not_same(expected, actual, [msg])
- assert_nil(obj, [msg])
- assert_not_nil(obj, [msg])
- assert_empty(obj, [msg])
- assert_not_empty(obj, [msg])
- ....

https://guides.rubyonrails.org/testing.html#available-assertions

Fixture

- Fixture nos permite hacer ejemplos de datos.
- En Rails, nos ayuda a crear objetos del modelo de manera fácil utilizando los mismos datos:
- Va en app/test/fixtures/nombre_del_modelo.yml:

```yaml
jp1:
  author: Juan P. Sandoval
  title: Ingenieria de Software
  year: 2023

jp2:
  author: Juan P. Sandoval
  title: Testing
  year: 2022
```

Unit test - Con fixture

```ruby
class Book < ApplicationRecord
    validates :title, presence: true
    validates :author, presence: true
    validates :year, presence: true
end
```

```ruby
class BookTest < ActiveSupport::TestCase
  test "should not save without title" do
    @book = books(:jp1)
    result = @book.save
    assert_not result,"saved the title without title"
  end
end
```

Integration test - Ejemplo

```ruby
class BooksControllerTest <
ActionDispatch::IntegrationTest
  test "should get index" do
    get books_url
    assert_response :success
  end

  test "should get new" do
    get new_book_url
    assert_response :success
  end

end
```

Envia un request GET a la url "/books" y verifica que el response sea success.

Envia un request GET a la url "/books/new" y verifica que el response sea success.

Integration test - Ejemplo

```
class BooksControllerTest < ActionDispatch::IntegrationTest
  test "should show book" do
    @book = Book.new(title:"Ing. Software",author:"Juan P.", year:2023)
    @book.save
    get book_url(@book)
    assert_response :success
  end

end
```

- Creamos un objeto libro en la base de datos
- Hacemos un GET request a /books/id, donde el id es el libro recién creado.
- Se verifica que el http response sea success.

Integration test - Ejemplo

```
test "should show book" do
    @book = Book.new(title:"Ing. Software",author:"Juan P.", year:2023)
    @book.save
    get "/books/#{@book.id}"
    assert_response :success
end
```

- También se puede armar la URL manualmente si no están acostumbrados a los path helpers (se pueden ver al hacer en terminal rails routes).

Integration test - Ejemplo

```
test "should create book" do
    assert_difference("Book.count") do
      post books_url, params:
           { book: { author: "Juan P.",
                     title: "Ing. Software",
                      year: 2022 } }
    end

    assert_redirected_to book_url(Book.last)
  end
```

Verifica que el contador incremento en uno

Verifica que se redirecciono a la url "books/id", siendo el id el del ultimo libro

Integration test - Ejemplo

```
test "should destroy book" do
    @book = Book.new(title:"Ing. Software",author:"Juan P.",
year:2023)
    @book.save
    assert_difference("Book.count", -1) do
      delete book_url(@book)
    end

    assert_redirected_to books_url
  end
```

Verifica que el contador decremento en uno

Verifica que se redirecciono a la url “books”, cuando se borra el controlador debería redireccionar a “/books”

Coverage

Simple COV

Es una gema que nos permite ver cuántas líneas hemos testeado, y cuántas no.

Para instalarlo hay que agregar la gema al archivo Gemfile, y ejecutar bundle install para que se instale:

```
gem 'simplecov', require: false, group: :test
```

Luego configurarlo en el archivo test/test_helper, o spec_helper.rb:

```
require 'simplecov'
SimpleCov.start

# el contenido anterior al archivo debe ir aqui abajo
```

Al ejecutar los tests

Simple Cov crea una carpeta coverage en el proyecto con un conjunto de HTML que muestra el resultado.
Solo hay que abrir el HTML en el browser:

app/controllers/books_controller.rb

92.59% lines covered

27 relevant lines. 25 lines covered and 2 lines missed.

```
1.  class BooksController < ApplicationController
2.
3.    # GET /books
4.    def index
5.      @books = Book.all
6.
7.    end
8.
9.    # GET /books/1
10.   def show
11.     @book = Book.find(params[:id])
12.
```

Líneas de código no cubiertas

```
25.   # POST /books
26.   def create
27.     @book = Book.new(book_params)
28.     if @book.save
29.       logger.info "redirect"
30.       redirect_to book_url(@book), notice: "Book was successfully created."
31.     else
32.       render :new, status: :unprocessable_entity
33.     end
34.   end
```

Creamos un test para la creación exitosa de un libro, pero no consideramos cuando un libro no puede ser creado.

Setup + Tear down

```ruby
class BooksControllerTest < ActionDispatch::IntegrationTest
  #codigo ejecutado antes de cada test
  setup do

  end

  # codigo ejecutado despues de cada test
  teardown do
    # normalmente es buena idea resetear el cache
    Rails.cache.clear
  end

end
```

https://guides.rubyonrails.org/testing.html

¿Consultas?

Ingeniería de Software

12 - Testing

IIC2143-3

Josefa España

jpespana@uc.cl