

IIC 2143 – Ingeniería de Software

# Testing

M. Trinidad Vargas  
mtvargas1@uc.cl

# Evaluación Temprana

Canvas > Encuesta de Docencia

**Reforzar lo que funciona bien**

**Cambiar lo que se puede mejorar**



# ¿Qué es testing?

“Testing es un proceso que consiste en un conjunto de actividades relacionadas con el ciclo de vida del software y otros productos para (i) determinar que satisfacen los requerimientos especificados, (ii) demostrar que son aptos para su propósito y (iii) detectar defectos.”

Foundations of Software Testing

ISTQB Certification

# ¿Por qué es necesario?

El software es **desarrollado por personas**

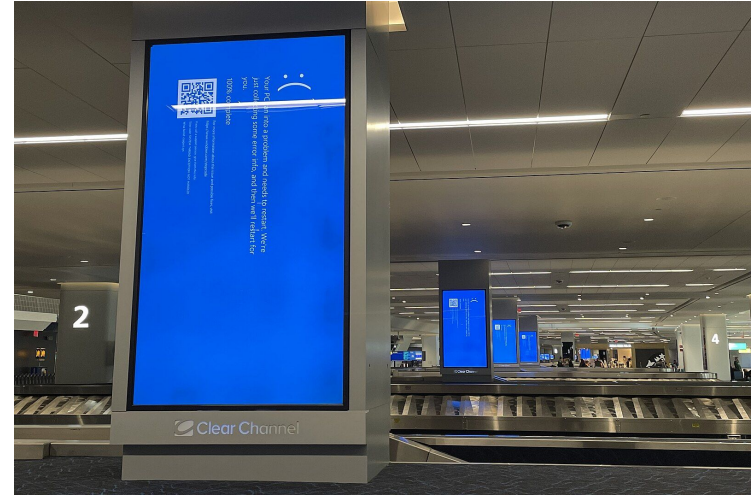
Es propenso a defectos

- Diseño
- Código
- Requerimientos
- Documentación, etc

# ¿Por qué es necesario?

Los defectos son costosos y pueden ser peligrosos

- Problemas con clientes
- Pérdidas monetarias
- Consecuencias fatales

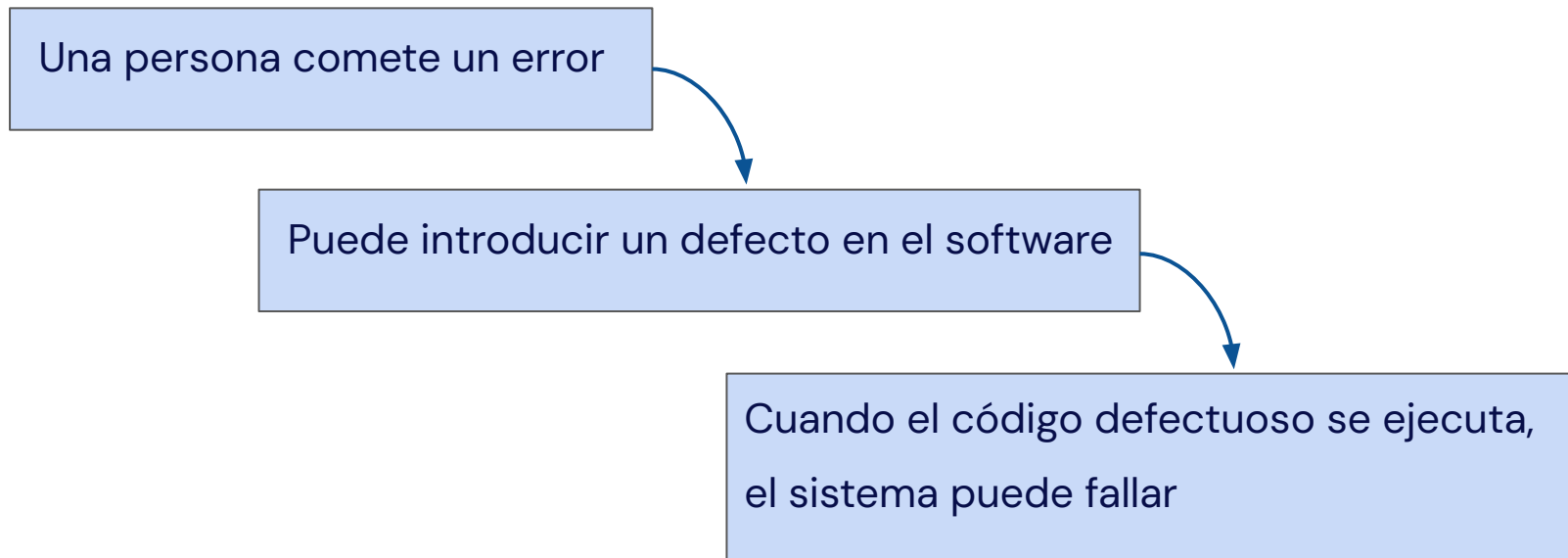


# ¿Por qué el software tiene defectos?

- **Error:** Acción humana que produce un resultado incorrecto
- **Defecto:** Presencia de una imperfección que puede ocasionar fallas.
- **Falla:** Comportamiento incorrecto observable de un componente o sistema con respecto a los requerimientos.

# ¿Por qué el software tiene defectos?

¿Cómo se relacionan los errores, defectos y fallas?



**¿Las LLM causan más errores?**



# ¿Las LLM causan más errores?

## Desventajas

- No tienen todo el contexto
- Suenan seguras aunque se equivoquen

**No necesariamente**, con el uso de LLM el rol del desarrollador cambia: pasamos a ser quienes guían, revisan y toman decisiones sobre el código, y el testing se vuelve aún más importante para validar lo generado.

# ¿Por qué el testing es importante?

- Cometemos errores
  - Detectar los errores de forma temprana facilita el desarrollo
  - Mientras antes se encuentran es más barato solucionarlos
- Queremos validar que todo sigue funcionando correctamente de forma rápida y confiable

# Beneficios del testing automatizado

- Aumenta la satisfacción del cliente
- Mejora la calidad del producto
- Reduce costos
- Previene catástrofes
- Acelera el desarrollo de software

# Test Driven Development

Es una metodología donde se escriben los tests antes que el código funcional

1. Escribir un test que falla (porque no existe la funcionalidad)
2. Escribir el código mínimo para que pase el test
3. Refactoriza el código sin romper el test

## Ventajas

- Obliga a pensar en los requisitos antes de escribir código
- Resulta código más limpio y enfocado
- Cobertura desde el día 1

# Tipos de testing

**Funcionales:** Evalúa que hace el sistema basado en requisitos

**No funcionales:** Evalúa cómo se comporta el sistema

- Pruebas de rendimiento
- Pruebas de carga
- Pruebas de usabilidad
- etc

**Nos enfocaremos en testing funcional**

# Niveles de testing

- **Pruebas Unitarias:** Prueban la unidad más pequeña del código como una función, método o clase. Funcionan correctamente aisladas del resto del código.
- **Pruebas de integración:** Verifica que varios módulos funcionan bien entre sí. Por ejemplo, los controladores.
- **Pruebas de sistema:** Evalúa la funcionalidad del sistema completa, incluyendo las vistas. Se abre la página web y se puede testear manualmente o automáticamente (bot).

# Testing en Rails

## ¿Dónde se implementan?

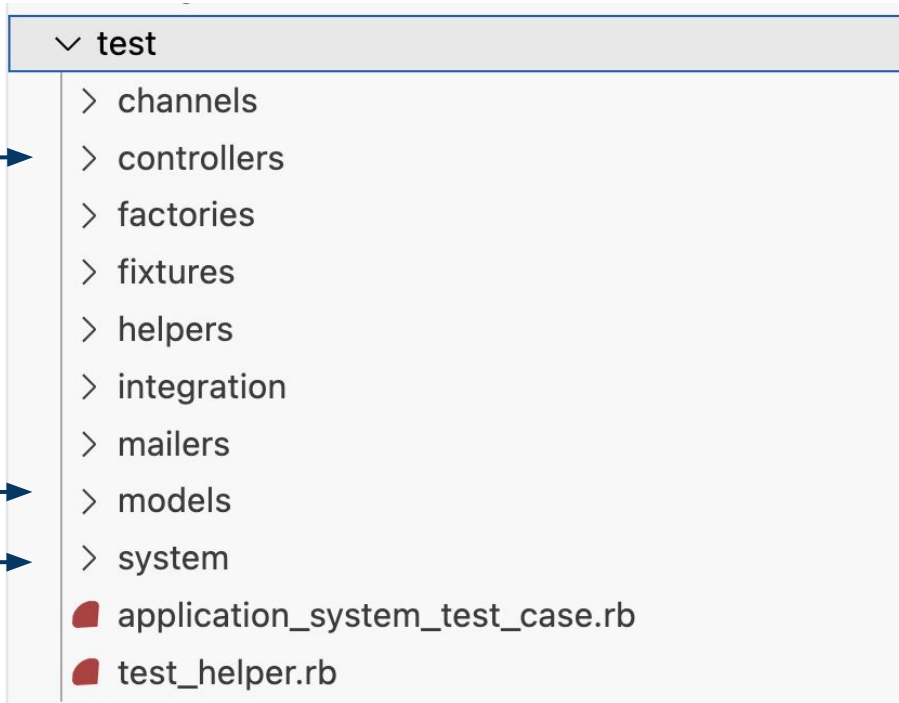
Test de controladores



Test de modelos



Test del sistema



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

# Ejemplo Unit Test de Modelo

app/models/recipe.rb

```
class Recipe < ApplicationRecord
  validates :name, presence: true
  validates :description, presence: true
end
```

test/models/recipe\_test.rb

```
class RecipeTest < ActiveSupport::TestCase
  test "Should not save without name" do
    recipe = Recipe.new
    result = recipe.save
    assert_not result, "Save Recipe without a name"
  end
end
```



# Estructura

```
class RecipeTest < ActiveSupport::TestCase
```

```
  test "Should not save without name" do
```

```
    recipe = Recipe.new
```

**Inicialización:** se crean los objetos y  
datos necesarios para el test

```
    result = recipe.save
```

**Estímulo:** donde se realizan la acción a testear

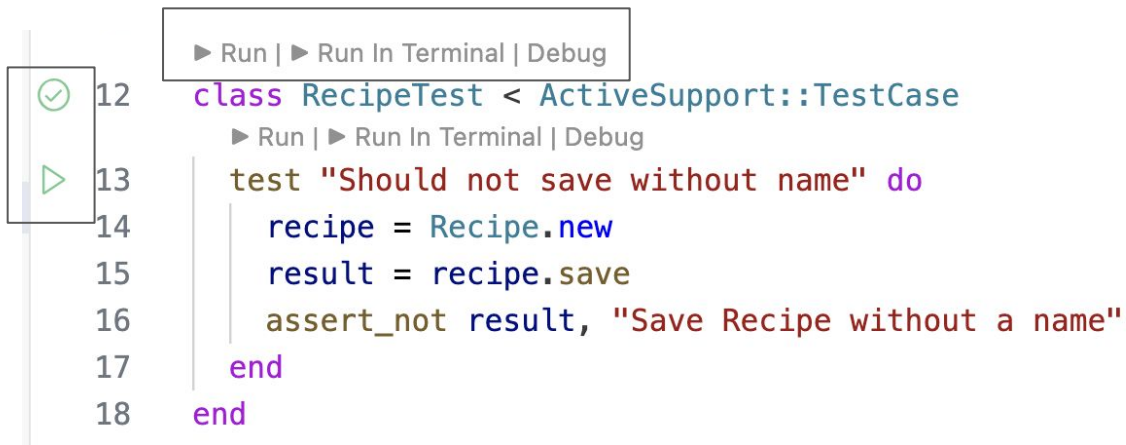
```
    assert_not result, "Save Recipe without a name"
```

```
  end  
end
```

**Verificación:** se evalúa si el resultado es el esperado

# Ejecutar los tests

```
> rails test
```



The screenshot shows an IDE window with a Ruby test file. On the left, a vertical toolbar contains a green checkmark icon and a green play button icon. The play button icon is highlighted with a white box. To the right of the toolbar, the code is displayed with line numbers 12 through 18. Above the code, a context menu is open, showing three options: 'Run', 'Run In Terminal', and 'Debug'. The 'Run' option is highlighted. The code itself is as follows:

```
12 class RecipeTest < ActiveSupport::TestCase
13   test "Should not save without name" do
14     recipe = Recipe.new
15     result = recipe.save
16     assert_not result, "Save Recipe without a name"
17   end
18 end
```

```
> rails test test/models/recipe_test.rb
```

```
Finished in 0.036745s, 27.2146 runs/s, 27.2146 assertions/s.
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

# Testing en Rails

- Rails tiene 3 bases de datos, uno para pruebas, otro para producción y otro para desarrollo. (Ver archivo config/database.yml).
- Rails inicializa una base de datos dedicada para ejecutar cada prueba, de esta forma cada test se ejecutará de forma separada y aislada.

# Asserts

- `assert( test, [msg])`
- `assert_not( test, [msg])`
- `assert_equal (expected, actual, [msg])`
- `assert_not_same (expected, actual, [msg])`
- `assert_nil ( obj, [msg] )`
- `assert_not_nil ( obj, [msg])`
- `assert_empty( obj, [msg])`

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

# Fixtures

Son archivos **predefinidos con datos de ejemplo**, que el sistema usa para poblar la base de datos durante los **tests**.

- Normalmente se escriben en formato yaml

```
test > fixtures > ! recipes.yml
1  recipe0:
2    name:
3    description:
4
5  recipe1:
6    name: "Empanada de queso"
7    description: "Empanadas fritas rellenas de queso"
8    preparation_time: 120
9    category: "Almuerzo"
10   servings: 6
```

<https://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html>

# Ejemplo Unit Test de Modelo

```
class RecipeTest < ActiveSupport::TestCase
  test "Should not save without name" do
    recipe = recipes(:recipe0)
    result = recipe.save
    assert_not result, "Save Recipe without a name"
  end
end
```

- Ahora usamos recipes(:recipe0)

# Ejemplo Integration Test de controlador

test/controllers/recipes\_controller\_test.rb

```
class RecipesControllerTest < ActionDispatch::IntegrationTest
  test "should get index" do
    get recipes_url
    assert_response :success
  end

  test "should get new recipe" do
    get new_recipe_url
    assert_response :success
  end
end
```

Realiza un request get a la url /recipes y verifica que la respuesta sea success

Realiza un request get a la url /recipes/new y verifica que la respuesta sea success

# Ejemplo Integration Test de controlador

```
test "should show recipe" do
  @recipe = Recipe.new(name: "Carrot Cake", description: "Best ...")
  @recipe.save

  get recipe_url(@recipe)
  assert_response :success
end
```

También se puede llamar al request con la url en vez del helper de path

```
get "recipes/#{@recipe.id}"
```



# Ejemplo Integration Test de controlador

```
test "should create recipe" do
  assert_difference("Recipe.count") do

    post recipes_url, params: { recipe: { name:
      "Blueberry Muffin", description: "A soft and
      fluffy muffin" } }
  end

  assert_redirected_to recipe_path(Recipe.last)
end
```

→ Verifica que el contador de recetas aumente en uno porque creamos uno con el post a recipes\_url

↙ Verifica que se redireccionó a la URL "recipes/id", usando el id del último artículo agregado.

# Ejemplo Integration Test de controlador

```
test "should destroy recipe" do
  @recipe = Recipe.new(name: "Test Recipe",
    description: "Will be destroyed")
  @recipe.save

  assert_difference("Recipe.count", -1) do
    delete_recipe_url(@recipe)
  end

  assert_redirected_to recipes_url
end
```

Verifica que el contador de recetas decrementa en uno

Verifica que se redireccionó a la URL `"/recipes"`.

# Coverage

Se refiere a la medición de qué tan bien están cubiertos tus archivos, funciones o líneas de código por los tests.

- En rails usaremos la gema **simplecov**

# Coverage

1. Agregamos simplecov al archivo Gemfile

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

2. Instalamos las nuevas gemas

```
> bundle install
```

3. Se agregan las siguientes líneas al principio del archivo

test/test\_helper.rb

```
require "simplecov"  
  
SimpleCov.start
```

4. Corremos los test igual que siempre

```
> rails test
```

# Reportes de simple-cov

Simple Cov crea una carpeta coverage en tu proyecto con un conjunto de archivos .html que reportan el resultado de la cobertura de código:

coverage

assets

.last\_run.json

.resultset.json

.resultset.json.lock

index.html

All Files ( 98.01% )

Generated less than a minute ago

All Files ( 98.01% covered at 2.37 hits/line )

28 files in total.  
403 relevant lines, 395 lines covered and 8 lines missed. ( 98.01% )

Search: app

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
app/controllers/reviews_controller.rb	86.67 %	59	30	26	4	1.70
app/controllers/books_controller.rb	94.87 %	65	39	37	2	1.44
app/controllers/authors_controller.rb	96.00 %	41	25	24	1	1.56
app/controllers/application_controller.rb	100.00 %	5	3	3	0	1.00
app/helpers/application_helper.rb	100.00 %	2	1	1	0	1.00
app/helpers/authors_helper.rb	100.00 %	2	1	1	0	1.00

# Reportes de simple-cov

El reporte nos informa que pedazos de código están siendo “cubiertos” (verde) por los tests que se ejecutaron y rojo “cubierta” (roja) por los tests actuales.

```
15. def update
16.   @review = Review.find(params[:id])
17.   if @review.update(review_params)
18.     render json: @review, status: :ok
19.   else
20.     render json: @review.errors, status: :unprocessable_entity
21.   end
22. end
```

# Set Up y Tear Down

Rails permite definir código que se ejecuta antes de cada test (setup) y código que se ejecuta después de cada test (teardown).

```
class RecipesControllerTest < ActionDispatch::IntegrationTest
  # código ejecutado antes de cada test
  setup do

  end

  # código ejecutado después de cada test
  teardown do
    Rails.cache.clear
  end
end
```



Es buena idea borrar el cache