.UBAfiuba
FACULTAD DE INGENIERÍA

Taller de Programación I

Testing en Rust

Ing. Pablo A. Deymonnaz - Ing. Martín Miletta
pdeymon@fi.uba.ar - mmiletta@gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

Índice

1. Unit testing
    - Qué es el Unit testing?
    - Unit tests en Rust
2. Integration tests
3. Estructura de un Test Unitario
4. Test Doubles / Mocking
5. Cobertura de Tests

Qué es el Unit testing?

Se define como un tipo de testing en el cual se prueba una "unidad" de software en forma individual o aislada.

- Ayudan a entender cual es la funcionalidad provista por nuestro código y como se utiliza.
- Permiten al programador realizar cambios en el código y verificar que continua funcionando correctamente.
- Hacen posible probar nuestro componente aislado, desacoplado del resto del programa.

Unit tests en Rust

En Rust los unit tests se escriben en el mismo archivo de código fuente.

- Los test se organizan en un módulo **test**.
- Se identifica a las funciones test con la anotación **#[test]** antes de la línea *fn*.
- Para realizar las afirmaciones, se utilizan las macros de la familia *assert*:
  - assert!(persona.edad > 18);
  - assert_eq!(4, 2+2);
  - assert_ne!(8, 3*4);
- Pueden usar atributos modificadores: **#[should_panic]**, **#[ignore]**

Tests que devuelven Result<T, E>

En Rust los unit tests pueden utilizar Result<T, E> como tipo de respuesta.

- Devuelven **Ok** si el test ejecuta correctamente.
- Devuelven **Err** cuando el test falla.
- Nos permiten utilizar el operador **?** en el cuerpo del test, lo que resulta una manera conveniente de hacer que el test falle cuando cualquier operación que se ejecuta devuelve un **Err**.
- No debemos utilizar **#[should_panic]**, en su lugar podemos hacer: **assert!(value.is_err())**.

Doc Tests

En Rust los ejemplos de documentación se ejecutan con los tests.

- Nos permiten asegurar que los ejemplos se mantienen actualizados.
- Se escriben utilizando triple comilla invertida (‘) al inicio y fin.
- Se consideran correctos si compilan y se ejecutan sin dar panic.
- Pueden hacer uso de los atributos: **should_panic**, **ignore**, etc.

Índice

1. Unit testing

2. Integration tests
   - Integration tests
   - Integration tests en Rust

3. Estructura de un Test Unitario

4. Test Doubles / Mocking

5. Cobertura de Tests

Integration tests

Prueban la funcionalidad de nuestro programa o biblioteca en forma integral, haciendo uso de la interfaz pública.

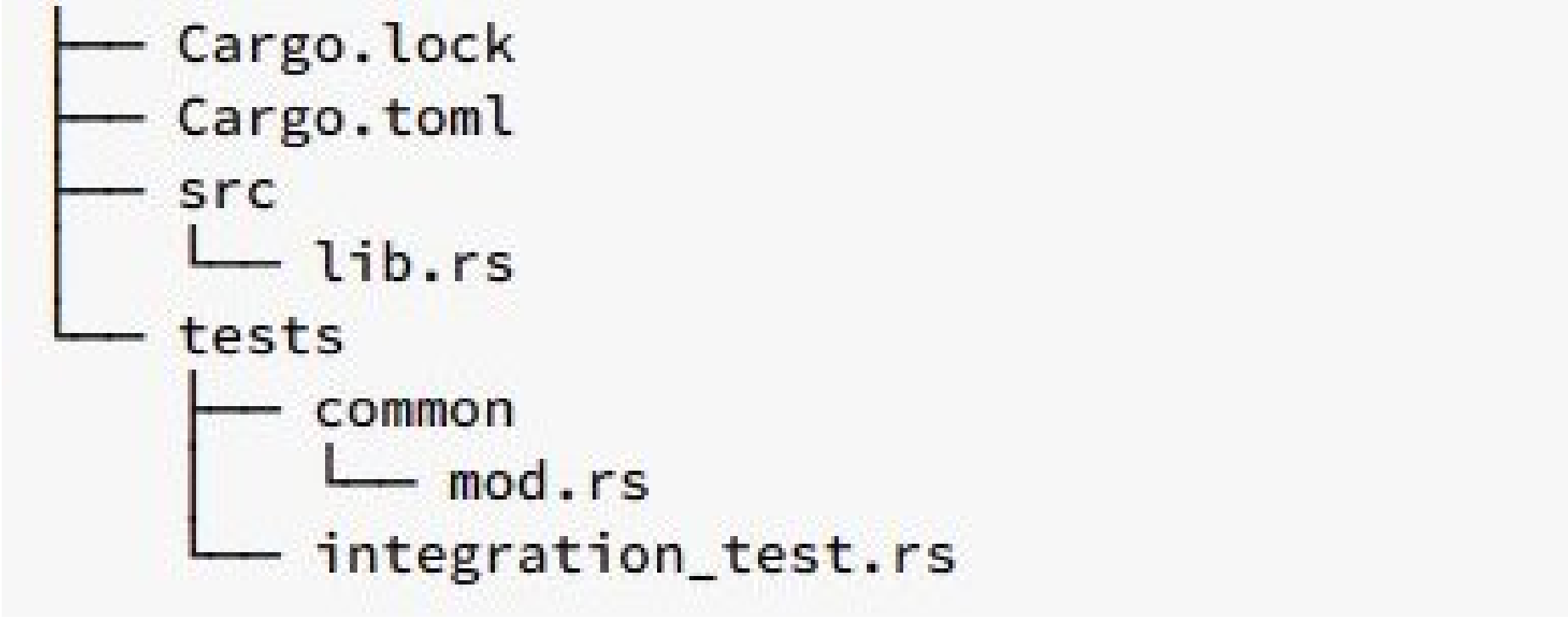
- Garantizan la funcionalidad completa del programa o biblioteca.
- No requieren aislar componentes individuales.
- En caso de fallar puede ser complicado encontrar donde esta el problema.
- Verifican que las distintas "unidades" de código funcionan adecuadamente en su conjunto (integradas).

Integration tests en Rust

- Se colocan en el directorio **tests**/, al lado de **src**/.
- Se compila cada archivo como un crate separado. Debemos incluir como crate nuestro código.
- Podemos compartir código entre test mediante el uso de subcarpetas.
- No pueden testear funciones de **src**/**main.rs**.

Organización de los Integration tests

Estructura de carpetas para organizar los integration tests:



.UBAfiuba FACULTAD DE INGENIERÍA

Índice

1. Unit testing
2. Integration tests
3. Estructura de un Test Unitario
4. Test Doubles / Mocking
5. Cobertura de Tests

Estructura de un Test Unitario

Hay varios patrones como: Given-When-Then (GWT) o Arrange-Act-Assert (AAA), que comparten la misma idea general de dividir el test en tres partes:

- Inicializar los datos y/o estado necesario para el escenario de prueba.
- Ejecutar el código que queremos probar.
- Verificar que los resultados son correctos (comparar con el resultado esperado).

.UBAfiuba FACULTAD DE INGENIERÍA

Índice

1. Unit testing
2. Integration tests
3. Estructura de un Test Unitario
4. Test Doubles / Mocking
5. Cobertura de Tests

Test Doubles / Mocking

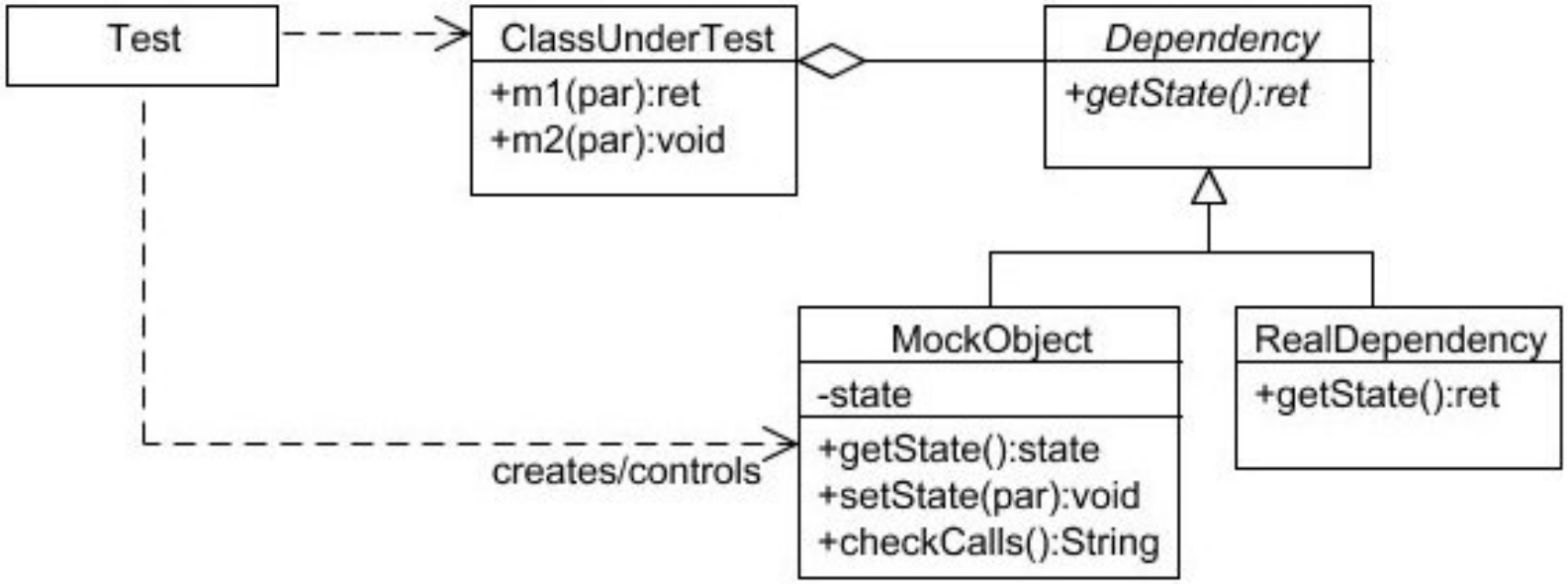
Es el procedimiento por el cual un componente externo del que depende nuestro código es reemplazado por otro que simula su comportamiento.

Al inicializar nuestro test podemos definir qué comportamiento queremos que tenga este componente externo simulado.

Nos permite aislar la "unidad" que queremos testear y definir distintos escenarios o casos de prueba.

También se pueden usar para verificar que funciones del componente fueron utilizadas y con qué parámetros.

Mocking - Diagrama UML



Índice

1. Unit testing
2. Integration tests
3. Estructura de un Test Unitario
4. Test Doubles / Mocking
5. Cobertura de Tests

Cobertura de Tests (code coverage)

Es una herramienta que nos permite observar qué porción de nuestro código esta siendo utilizado al ejecutar los tests.
Nos da métricas como las siguientes:

- Porcentaje de lineas ejecutadas por archivo fuente.
- Porcentaje de funciones o métodos ejecutados.
- Porcentaje de ramas condicionales ejecutadas (como bloques if-else).
- Cantidad de veces que se ejecutó cada línea de código.

Reporte de Cobertura (ejemplo)

Para obtener el code coverage en Rust podemos utilizar:

- cargo llvm-cov --html

Coverage Report

Created: 2022-10-15 18:15

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
basic.rs	100.00% (10/10)	100.00% (32/32)	90.48% (19/21)	- (0/0)
doc.rs	0.00% (0/2)	0.00% (0/10)	0.00% (0/4)	- (0/0)
files.rs	100.00% (7/7)	93.44% (57/61)	82.22% (37/45)	- (0/0)
lib.rs	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)	- (0/0)
server.rs	90.00% (9/10)	95.95% (71/74)	84.21% (32/38)	- (0/0)
Totals	**90.00% (27/30)**	**90.45% (161/178)**	**81.65% (89/109)**	**- (0/0)**

Generated by llvm-cov -- llvm version 14.0.5-rust-1.63.0-stable

Bibliografía

- **The Rust Programming Language**,
  Chapter 11: Writing Automated Tests
  Chapter 15: Interior Mutability
  https://doc.rust-lang.org/book/
- **Rust By Example**, Chapter 21: Testing.
  https://doc.rust-lang.org/rust-by-example/testing.html
- **Unit Testing Principles, Practices, and Patterns**, Vladimir Khorikov.
  Chapter 3 The anatomy of a unit test
  Chapter 5 Mocks