RUST
Yazılım Testi
Unit tests
Integration tests

Test Nasıl Yazılır?

Testler, program kodunun beklenen şekilde çalıştığını doğrulayan fonksiyonlardır. Bu fonksiyonlarda;

- gerekli veriler ve durum ayarlanır
- test edilecek kod çalıştırılır
- sonuçlarının beklenildiği şekilde olup olmadığı kontrol edilir

Yeni bir library create oluşturulduğunda, test kısmı projeye otomatik eklenir.

cargo new test-workshop --lib

```rust
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

cargo test → ile test çalıştırılır

#[cfg(test)] → modülün test modülü olduğun belirten özelliktir. Kodlar derlendiğinde binary kısma eklenmemesi sağlanır.

#[test] → fonksiyonun test fonksiyonu olduğunu belirtir.

Modül içinde test için kullanılmayan, bazı ortak işlemleri ve senaryoları gerçekleştiren fonksiyonlarda eklenebilir.

assert_eq!() makrosu → Verilen argümanlarda eşitlik arar. Bu örnekte fonksiyonun gönderdiği değerin 4 e eşit olmasını kontrol eder. Aranan eşitlik oluşmaz ise test başarısız olur.

use süper::* → Test molülü dışındaki bütün öğeler ulaşılabilir hale gelir.

cargo test --> komutuna argümanda ekleyebiliriz. Bu işleme filtreleme adı verilir. (sonraki konularda göreceğiz)

measured → Performansı ölçen, kıyaslama testleri içindir. (https://doc.rust-lang.org/unstable-book/library-features/test.html)

Doc-tests .... → (https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html#documentation-comments-as-tests)

Başarısız Test :

```
#[test]
fn another() {
    panic!("Make this test fail");
}
```

assert!() Macrosunun Kullanımı

Makroya koşullu argümanlar verilir
Sonuç true ise test başarılı olur, false ise panic! makrosu çalışır ve test başarısız olur.

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Testler (assert!())

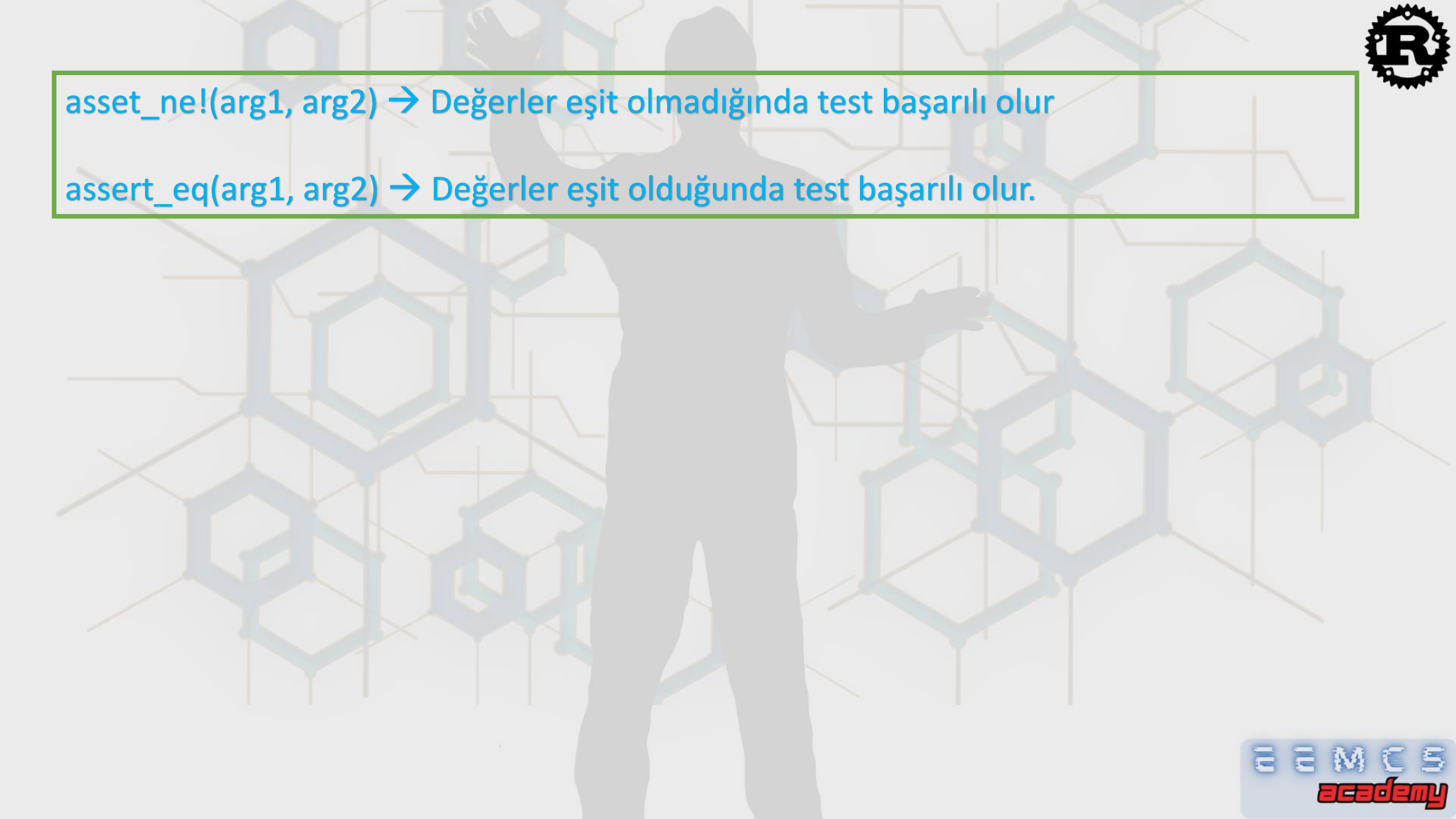
```
#[test]
fn larger_can_hold_smaller() {
    let larger = Rectangle {
        width: 8,
        height: 7,
    };
    let smaller = Rectangle {
        width: 5,
        height: 1,
    };

    assert!(larger.can_hold(&smaller));
}
```

```
fn smaller_cannot_hold_larger() {
    let larger = Rectangle {
        width: 8,
        height: 7,
    };
    let smaller = Rectangle {
        width: 5,
        height: 1,
    };

    assert!(!smaller.can_hold(&larger));
}
```

Başarısız olması için fonksiyonda ">" birini değiştirebiliriz.

asset_ne!(arg1, arg2) → Değerler eşit olmadığında test başarılı olur

assert_eq(arg1, arg2) → Değerler eşit olduğunda test başarılı olur.



Başarısızlık Mesajını Özelleştirme

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

```
    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(
            result.contains("Carol"),
            "Greeting did not contain name, value was `{}`",
            result
        );
    }
```

panic!() Makrosunun #[should_panic] ile Kontrol Edilmesi

Hata koşullarının beklediğimiz gibi olup olmadığını kontrol etmek içinkullanılır.
Bunun için should_panic özelliğini eklememiz gerekir.
Panic olması için gerekli koşul oluştuğunda hata olursa test başarılı olur, hata olmaz ise test başarısız olur.

```rust
pub struct Guess {
    value: i32,
}
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Test başarılı olur.

```rust
pub struct Guess {
    value: i32,
}
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1  {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Programa bug eklersek Test BAŞARISIZ olur.

Should_panic kullanan testler kesin olmayabilir.
Should_panic testi, beklemediğimiz bir nedenle panik olsa bile testi geçebilir.

Daha kesin hale getirmek için parametre ekleyebiliriz.

```
#[should_panic(expected = "less than or equal to 100")]
```

expected → panic mesajını kontrol ederek testi daha sağlamlaştırabiliriz. Eğer oluşan panic mesajı içinde expected kısmında belirtilen mesaj yoksa test başarısız olur, varsa test başarılı olur.

```rust
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
                "Guess value must be greater than or equal to 1, got {}.",
                value
            );
        } else if value > 100 {
            panic!(
                "Guess value must be less than or equal to 100, got {}.",
                value
            );
        }

        Guess { value }
    }
}
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Test başarılı olur

```rust
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
                "Guess value must be less than or equal to 100, got {}.",

                value
            );
        } else if value > 100 {
            panic!(
                "Guess value must be greater than or equal to 1, got {}.",
                value
            );
        }

        Guess { value }
    }
}
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Test BAŞARISIZ olur

Test İçinde Result<T, E> Kullanımı

Test başarılı olunca Ok(()), başarısız olunca Err(...) döner.
? Opeatörü kullanılabilir.
Herhangi bir işlem Err varyantını gönderirse test başarısız olur.

```
#[test]
fn it_works() -> Result<(), String> {
    if 2 + 2 == 4 {
        Ok(())
    } else {
        Err(String::from("two plus two does not equal four"))
    }
}
```

Testlerin Çalışmasını Kontrol Etmek :

Cargo test ;

- kodu test modunda derler
- oluşan binary kodu çalıştırır

Bu aşamada;

- tüm testleri paralel olarak çalıştırmak
- test çalıştırmaları sırasında üretilen çıktıyı yakalamak
- çıktının görüntülenmesini engellemek

gibi işlemlerle testlerin çalışma durumunu kontrol edebiliriz.

Bunun için komut satırı seçenekleri de kullanılabilir.

cargo test --help cargo test -- --help

Testleri Paralel yada Ardışık Çalıştırmak

```
$ cargo test -- --test-threads=1
```

Testler paralel çalışmaz. 1-bir değeri ile de sadece bir adet iş parçasının çalıştırılmasını belirtir.

Fonksiyon Çıktısını Göstermek

Test başarılı olunca, standart çıktıya (println! İle yazdırılanlar) yazdırılan herşeyi yakalar fakat test ekranında gözükmez;
Test başarısız olunca, görürüz.

Bütün mesajları görmek için testi aşağıdaki gibi çağırabiliriz.

```
$ cargo test -- --show-output
```

İsme Göre Testin Bir Alt Kümesini Çalıştırmak

cargo test test-fonksiyonu-adı

Filtreleme İle Birden Fazla Test Çalıştırmak

cargo test test-fonksiyonu-adının-belli-bir-kısmı

Cargo test kısmında verilen değerin, test fonksiyonlarının isminde bulunan bütün testler çalışır.

Bazı Testlerin Devre Dışı Bırakılması

#[ignore] → özelliği eklenen testler devre dışı kalır.

cargo test -- --ignored → Sadece devre dışı bırakılan testler çalışır.

cargo test -- --include-ignored → Devre dışı olanlarıda dahil eder.

Testlerin Organizasyonu

Rust community;

- unit tests
- integration tests

olmak üzere iki ana kategoriye ayırır.

Unit tests → Birim testleri küçük ve daha odaklıdır, her seferinde tek bir modülü izole olarak test eder ve özel arayüzleri test edebilir (ŞİMDİYE KADAR YAPTIĞIMIZ ÖRNEKLER)

Integration tests → kütüphanenizin tamamen dışındadır ve kodunuzu diğer harici kodlarla aynı şekilde kullanır, yalnızca genel arayüzü kullanır ve potansiyel olarak test başına birden fazla modülü çalıştırır.

Integration Tests

Kütüphanenizin birçok parçasının birlikte doğru çalışıp çalışmadığını test etmektir.

tests → adında bir dizin oluşturulur kök klasörde

```
adder
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

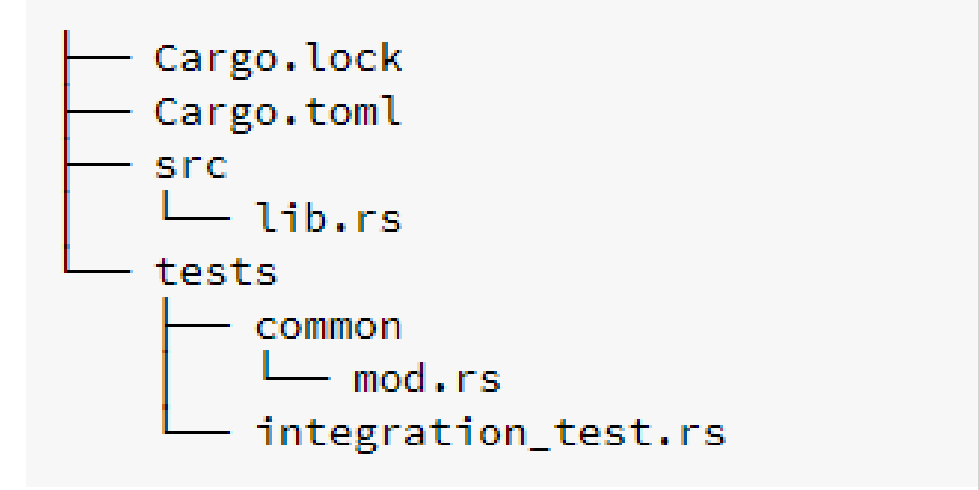
```rust
use adder;// sizin oluşturduğunuz lib projesinin adı olarak değiştirmeyi unutmayın

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

Use ile kütüphane içe aktarılır ve testler yazılır.

Cargo test → ile çalışır

cargo test --test test-dosyası-adı → Belli bir test dosyası çalıştırılır

Integration Testlerde Alt Modül Oluşturma



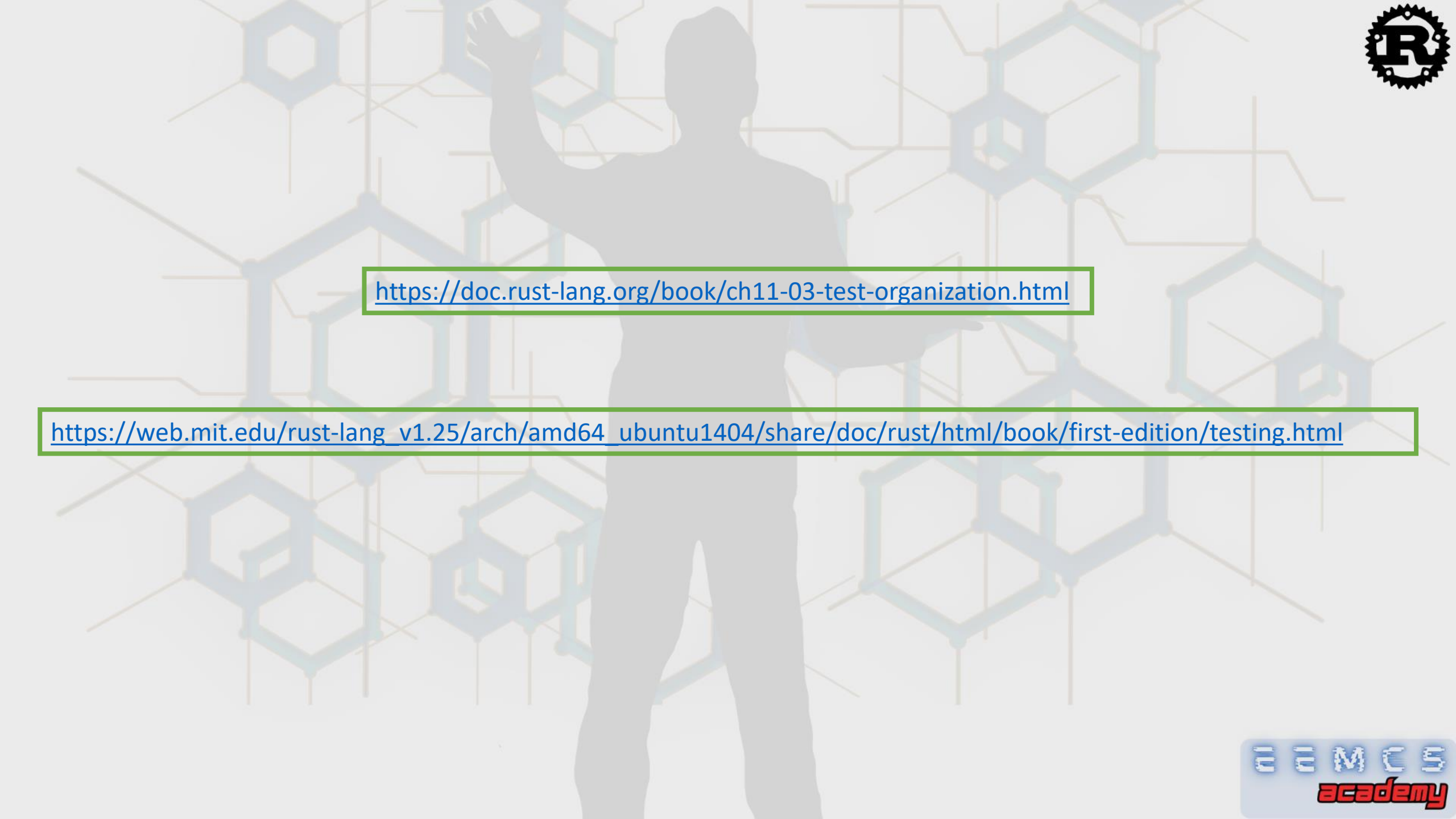

```rust
use adder;

mod common;

#[test]
fn it_adds_two() {
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

Binary Crate'ler İçin Integration Testler

Projemiz yalnızca bir src/main.rs dosyası içeren ve src/lib.rs dosyası içermeyen bir binary crate ise, tests dizininde entegrasyon testleri oluşturamaz ve src/main.rs dosyasında tanımlanan işlevleri bir use deyimi ile kapsama alamayız.

Yalnızca kütüphane crate'leri diğer crate'lerin kullanabileceği fonksiyonları açığa çıkarır; binary crate'ler kendi başlarına çalıştırılmak üzere tasarlanmıştır.

Bu, binary crate bulunan Rust projelerinin src/lib.rs dosyasında bulunan mantığı çağıran basit bir src/main.rs dosyasına sahip olmasının nedenlerinden biridir.



https://doc.rust-lang.org/book/ch11-03-test-organization.html

https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/testing.html

Celal AKSU
Bilişim Teknolojileri Öğretmeni

celalaksu@gmail.com

https://www.linkedin.com/in/cllaksu/

https://twitter.com/ksacll

https://www.youtube.com/@eemcs