

Systemnahe Programmierung in Rust

- "The Book" / Automatisierte Tests schreiben / Kap. 11 -

Hubert Högl

Technische Hochschule Augsburg / Informatik
<https://tha.de/~hhoegl>

2024-10-05 13:23:59

Warum Tests?

- Durch Tests kann man das Vorhandensein von Fehlern zeigen.
- Rust kann bereits beim Kompilieren viele Fehler im Code finden, allerdings keine Logikfehler.
- Tests können Logikfehler finden.
- Es können Funktionen, Module (Sammlung aus privaten und öffentlichen Funktionen) und die öffentliche Schnittstelle von grössere Einheiten (Integrationstests) getestet werden.

Tests schreiben

Der Rumpf von Testfunktionen führt in der Regel diese drei Aktionen aus:

- Bereite die benötigten Daten und Zustände vor.
- Führe den Code aus, den du testen möchtest.
- Stelle sicher, dass die Ergebnisse das sind, was du erwartest.

```
$ cargo new adder --lib
```

Die Datei lib.rs:

```
#[cfg(test)]
mod tests {
    #[test]    // <-- Annotation: Testfunktion
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

```
$ cargo test
```

Tests schreiben (2)

Ausgabe

```
running 1 test
```

```
test tests::it_works ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

```
Doc-tests adder
```

```
running 0 tests
```

```
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

- passed/failed
- ignored: Tests können als ignoriert markiert werden (`#[ignore]`)
- measured: Benchmark Tests (erst in Zukunft möglich)
- Doc-tests: Dokumentationstests (Kap. 14.2.)

Tests schreiben (3)

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() { // fails!
        panic!("Lasse diesen Test fehlschlagen");
    }
}
```

- `assert!`, `assert_eq!`, `assert_ne!`
- `assert!(x, "x wasn't true!");`
- `assert_eq!(a, b, "we are testing addition with {} and {}", a, b);`
- Damit `assert!` klappt: `#[derive(PartialEq, Debug)]`

```
use super::*;
```

Tests schreiben (4)

should_panic

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

Tests können auch `Result<T, E>` zurückgeben, statt das Ergebnis mit `assert!` zu prüfen.

Tests steuern

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

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

```
$ cargo test one_hundred    # nur Test `one_hundred` laeuft
```

```
$ cargo test add    # alle Tests add... laufen (add_two_and_two, add_three_and_two)
```

```
$ cargo test -- --ignored    # Ausfuehren der ignorierten Tests
```

```
cargo test -- --include-ignored
```

- Modultests (unit tests)
- Integrationstests (integration tests)

Modultests

Wird nur mit cargo test kompiliert.

```
#[cfg(test)] // Konfigurationsoption 'test'  
mod tests { // Testmodul  
    ...  
}
```

Private Funktionen (ohne pub) kann man testen oder auch nicht.

Integrationstest

Geht nur bei Bibliotheken.

In Verzeichnis tests (neben src).

tests/integration_test.rs (eigenes Crate):

```
use adder;
```

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

Bei cargo test laufen nun: 1) Modultests, 2) Integrationstests, 3) Dokumentationstests.

Mehr Test-Crates können in tests/ hinzugefügt werden.

Bestimmten Test laufen lassen:

```
$ cargo test --test integration_test
```