17 Testen

Zu moderner Entwicklung robuster Software gehören automatisierte Tests. Die Definition von Test-Driven Development oder gar eine Diskussion um die Abgrenzung dieses Ansatzes wollen wir uns an dieser Stelle sparen. Vielmehr wollen wir uns Problemstellungen aus der Praxis konkreter Umsetzungen anschauen.

Vorausschicken wollen wir zudem, dass Tests keine Garantie für fehlerfreie Software geben. Tests stellen im Idealfall sicher, dass eine Funktionalität, ein Verhalten auch bei Änderung der Software weiterhin bestehen bleibt – nicht mehr, aber auch nicht weniger.

17.1 Arten von Tests

Bevor wir uns die Entwicklung konkreter Tests in Rust anschauen, wollen wir versuchen, eine Definition und Abgrenzung zu finden.

Viele Testbegriffe

Die bekannteste Art der automatisierten Tests sind *Unit-Tests*. Wahrscheinlich haben Sie diesen Begriff bereits gehört und auch schon Unit-Tests geschrieben. Weitere Begriffe wie *Systemtest*, *Akzeptanztest*, *Oberflächentest*, *UI-Test*, *Komponententest*, *Regressionstest* begegnen uns im Alltag. Diese wollen wir zunächst einsortieren. Um das zu erreichen, beginnen wir mit einem Blick auf die Testpyramide.

Die Testpyramide hat sich in der Softwareentwicklungsbranche etabliert und wird auch von uns als Referenz herangezogen. Diese zeigt zum einen auf, welche grundlegenden Arten von Tests existieren und zum anderen, wie groß die Anzahl an geschriebenen und zur Verfügung stehenden Tests in einem Softwareprojekt sein sollte. Die Testpyramide unterscheidet drei Arten von Tests.

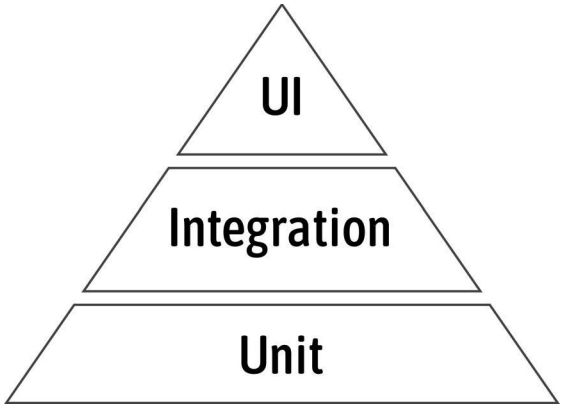

Abb. 17–1
Testpyramide

17.1.1 Unit-Tests

Von unten gesehen beginnen wir mit Unit-Tests. Bei diesen geht es darum, eine Unit (Einheit) aus ihrem produktivem Einsatzgebiet herauszuziehen (Isolierung) und das erwartete Verhalten dieser Einheit zu überprüfen.

Ein Unit-Test schmiegt sich im Idealfall an den produktiven Code derart an, dass aus dem Test der produktive Code beinahe vom Test abgeleitet werden kann. Er ist wie eine Negativform bei Gipsarbeiten oder besser ausgedrückt wie eine Backform, in der ein speziell geformter Kuchen, wie zum Beispiel ein Gugelhupf, gebacken werden soll.

Tipps und Tricks

Unit-Tests sind wie Backformen, mit denen produktiver Code gebacken werden könnte.

Um die eben erwähnte Isolierung zu erreichen, versorgen wir den produktiven Code innerhalb der Tests mit immer gleichen Daten und gegebenenfalls auch mit immer dem gleichen Programmcode, den unsere Unit zum Laufen braucht. So stellen wir sicher, dass bei jedem Testlauf derselbe Zustand herrscht und das Testergebnis nicht verfälscht wird. Salopp ausgedrückt: Wir nehmen immer die gleiche Backform und immer die gleiche Menge an Butter, sodass wir auch immer das gleiche Ergebnis erhalten.

Durch die oben erwähnte Isolierung gewinnen wir mehrere Vorteile für unsere Tests. Zum einen können wir uns auf nur einen (kleinen) Teil unseres Systems konzentrieren. Zum anderen können wir bei einem fehlschlagenden Test sehr schnell feststellen, an welcher Einheit sich eine Änderung ergeben hat und unserer Aufmerksamkeit bedarf.

In der Praxis sind wir oft darauf angewiesen, den Programmcode, von dem unser produktiver Code abhängig ist, durch einen Mock zu ersetzen. Um bei unserem Beispiel des Backens zu bleiben: Es müssen nicht unbedingt echte Schokostücke aus Zartbitter in den Teig hinein. Zum Füllen der Form können es auch Glasmurmeln sein. Hauptsache ist, sie verteilen sich im Teig, füllen mit ihm zusammen die Form aus, und wir können das Gesamtbild nach dem Stürzen begutachten. Mehr dazu beschreiben wir in Abschnitt 17.4.

17.1.2 Integration-Tests

Nachdem wir mit Unit-Tests einzelne Stücke isoliert und getestet haben, kombinieren wir diese im nächsten Schritt miteinander und überprüfen, ob diese Kombination auch nach unseren Vorstellungen funktioniert. Wir bringen somit den Teig mit dem Schokoguss und den Schokolinsen zusammen. Unter diesem Test verstehen wir den Integration-Test.

Bei dieser Definition könnten wir uns fragen, welche einzelne Stücke wir nun kombinieren. Sind es einzelne Klassen, einzelne Module oder einzelne Crates? Wo fängt der Integration-Test an und wo hört er auf? Unserer Ansicht nach sind alle Tests – unabhängig von der Integration – eben Integration-Tests. Es ist aus unserer Sicht unerheblich, welche Einzelstücke zusammengefügt werden. Viel wichtiger sollte uns sein, dass wir bei einem Integration-Test ausschließlich das Zusammenspiel und damit die Protokolle (API, Funktionssignaturen, Konstruktoren und so weiter) der betrachteten Einzelstücke testen. Dabei sollten wir darauf achten, das erneute Testen der Einheit selbst zu vermeiden.

Tipps und Tricks

Integration-Tests testen ausschließlich die Verbindung und damit die Protokolle (Schnittstellen) der Einzelstücke (Units).

Schauen wir noch einmal auf die Testpyramide, so stellen wir fest, dass die Integration-Tests in dieser weniger Platz einnehmen. Das hat den Grund, dass wir beim Testen der Einzelstücke in der Regel deutlich mehr Tests schreiben als beim Testen der Verbindungen oder Zusammenschaltungen. Somit fällt die Anzahl der Integration-Tests auch niedriger aus als die der Unit-Tests.

Diese Beobachtung der Testpyramide wird in der Softwaregemeinschaft des Öfteren diskutiert. Uns begegneten in der Praxis Ansätze,

durch den Integration-Test alle Einheiten mit zu testen und dadurch den Aufwand zu sparen. Wie wir aber weiter oben schon ausgeführt haben, führt dieser Ansatz in der Praxis zu einer längeren Suche des Fehlers/der Änderung, da der Test viele einzelne Einheiten mit abdeckt und wir diese erst einmal identifizieren müssen. Meistens passiert diese Suche auch zu einem Zeitpunkt, an dem wir es gerade überhaupt nicht gebrauchen können. Insofern können wir nur dazu raten: Die Integration-Tests sind ein Zu- und kein Ersatz.

Im Hinblick auf Rust werden wir sehen, dass Integration-Tests die Einzelstücke gar nicht mehr betrachten, sondern dass nur ein komplettes Crate an sich getestet werden kann. Dazu mehr in Abschnitt 17.1.2.

17.1.3 UI-Tests

Die Spitze und damit der kleinste Teil repräsentiert jenen Teil, über den Software-Entwicklungsteams sich immer wieder den Kopf zerbrechen. Das liegt an Fragen wie:

1. Wie sollen wir das denn testen?
2. Wie schaffen wir es, die Tests aktuell zu halten?
3. Was genau ist die UI?

Auch hier versuchen wir pragmatische Antworten zu finden und möchten Ihnen folgenden Leitfaden für UI-Tests vorschlagen:

4. Die UI ist das, womit der Benutzer interagiert. Das kann das dargestellte HTML im Browser bei einer Webanwendung, es können aber auch die verschiedenen Schalter (engl. *switches*) bei einer Konsolenanwendung (CLI = Command Line Interface) sein.
5. UI-Tests sollten lediglich als Ergänzung zu den Unit- und Integration-Tests dienen.
6. UI-Tests testen nur die kritischsten Bereiche. UI-Tests sind fragil, da sie durch eine kleine Änderung im System fälschlicherweise fehlschlagen – und damit ein *false negative* auslösen – können. Daraus folgt auch, dass wir eher weniger als zu viele UI-Tests empfehlen.

End2End-Tests

In der Literatur und im Web finden wir auch die Bezeichnung Ende-zu-Ende-Test (oder auch End2End, E2E). Wir fassen diese Bezeichnung als Synonym auf. Wichtig bei dem Thema ist uns weniger die Bezeichnung als vielmehr das Ziel hinter und damit die jeweilige Abdeckung der UI- oder E2E-Tests. Es geht um einen Test des vollständigen Systems, in dem wir den Nutzer simulieren. Die Fragilität und die Komplexität bleiben davon unberührt.

Tipps und Tricks

Aufgrund der Fragilität und der hohen Spezifität von UI-Tests sollten wir uns darauf konzentrieren, mit UI-Tests wenige, geschäftskritische Fälle abzudecken. Haben wir die Möglichkeit, ein Zusammenspiel mehrerer Systemkomponenten über Integration-Tests abzudecken, sollten wir diese in jedem Fall UI-Tests vorziehen.

17.1.4 Testpyramide, Nachwort

Die Testpyramide ist umstritten. Von ihr existieren auch Abwandlungen, wie zum Beispiel die invertierte (also um 180° gedrehte Variante) oder ein Testdiamant (viele Integration-Tests und sonst wenig andere Tests). Diese Abwandlungen werden als Alternative oder als etwas, was wir unbedingt vermeiden sollten (Anti-Pattern), vorgestellt. Des Öfteren beschreibt die Testpyramide auch die angestrebte Theorie, während die invertierte Variante die Realität widerspiegelt. Somit gibt es das Ziel, viele Unit-Tests zu schreiben, aber aufgrund von Zeitdruck, fehlendem Wissen oder organisatorischen Konflikten wird doch das meiste manuell getestet. Dieses Thema hinreichend zu beleuchten sprengte den Rahmen dieses Buchs. Wir wollen dennoch festhalten, dass die Idee der Dreifaltigkeit an Tests – und ihre Häufigkeit an der Testpyramide abzulesen – aus unserer Sicht sinnvoll und anzustreben ist.

17.2 Rust, Cargo und Tests

Rust und Cargo kennen von Haus aus nur Unit- und Integration-Tests und unterscheiden diese strikt. So liegen die Unit-Tests direkt bei der Unit im selben Modul. Sie werden laut Konvention als Untermodule vom Hauptmodul getrennt. Mehr dazu in Abschnitt 17.2.1.

Integration-Tests hingegen betrachten das Crate von außen und werden daher auch außerhalb des `src`-Verzeichnisses in einem separaten Verzeichnis namens `test` in einem Cargo-Projekt abgelegt.

17.2.1 Platzierung von Testcode

Unit-Tests

Mit jeder Änderung am Produktionscode sollte sich der Test anpassen – und umgekehrt. Insofern ist es eine gute Idee, den Testcode auch so nah wie möglich am produktiven Code zu platzieren. Durch die Möglichkeit, Module verschachteln (Modul in Modul) und diese mit Attributen markieren zu können, hat sich ein De-facto-Standard zur Platzierung des Unit-Test-Codes durchgesetzt.

Tipps und Tricks

Unit-Test-Code kommt in ein Submodul namens `tests`.

Schauen wir uns das kurz an:

Listing 17–1
Einfachstes Testbeispiel

```
pub fn hello() -> String {
    String::from("Hello")
}

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

    #[test]
    fn test_hello() {
        assert_eq!(hello(), "Hello");
    }
}
```

Wir definieren im ersten Schritt eine einfache Funktion `hello()`, welche die Zeichenkette `Hello` zurückgibt.

Weiter erstellen wir im aktuellen Modul – oder auch in der `main.rs` oder `lib.rs` – ein weiteres Modul mit dem Namen `tests`. Dieses markieren wir `#[cfg(test)]`. Damit wird der Quelltext im Modul `tests` für den produktiven Bau ignoriert und nur beim Aufrufen von `cargo test` aufgerufen. In diesem Modul importieren wir alle Elemente aus dem Supermodul, in unserem Fall dadurch lediglich `hello()`. Wenn wir uns dieses Import-Statement angewöhnen, haben wir für die zukünftigen Tests alle Elemente schon importiert. Wir können alternativ auch lediglich `use super::hello;` verwenden.

Welcher Weg der bessere ist, hängt nicht nur vom Geschmack, sondern auch von dem von uns verwendeten Editor ab. Bei der Verwendung einer IDE wie IntelliJ oder Eclipse sind einzelne Imports einfacher zu erreichen. Bei einem Text-Editor wie vim oder Sublime sind Asterisk-Imports sicher einfacher. Entscheiden Sie gerne selbst.

Die eigentliche Testfunktion schreiben wir als Nächstes und markieren diese mit dem Attribut `#[test]`. Den Namen der Funktion können wir frei wählen. Dazu später mehr in Abschnitt 17.3.2. Innerhalb der Funktion vergleichen wir mit dem Makro `assert_eq!` den Rückgabewert von `hello()` mit der Zeichenkette `Hello`. Auch auf die Assertions gehen wir später in Abschnitt 17.3.1 detaillierter ein. Für jetzt ist nur wichtig, dass wir mit diesen drei Schritten (Submodul `tests`, Attribut `#[test]` und Verwendung von `assert_eq!`) alles zusammengestellt haben, was wir für einen Unit-Test benötigen.

Integration-Tests

Nachdem wir den Code von Unit-Tests so nah wie möglich am produktiven Code platziert haben, drehen wir das für den Integration-Test um und platzieren ihn so weit weg wie möglich. Die Idee hierbei ist, dass der Integration-Test den produktiven Code so betrachtet, als wäre der Integration-Test ein separates Crate.

Angenommen wir erstellen ein Bibliothek-Crate namens `rustbuch_testing` und füllen die `lib.rs` mit folgender, öffentlich verfügbarer Funktion:

```
pub fn hello_outside_world() -> String{
    String::from("Hello, outside world!")
}
```

Listing 17–2
Einfache Funktion, die wir später mit einem Integration-Test testen wollen

Wir erstellen im Wurzelverzeichnis des Projektes ein Verzeichnis namens `tests`. Wir legen eine Datei an, die wir des einfachen Verständnisses halber `integration.rs nennen`. Die Datei füllen wir mit folgendem Code:

```
#[test]
fn test_hello_outside_world() {
    assert_eq!(rustbuch_testing::hello_outside_world(),
               "Hello, outside world!");
}
```

Listing 17–3
Integration-Test von hello_outside_world()

Wir rufen die oben definierte Funktion mit dem Präfix `rustbuch_testing::` auf, was dem Namen des Crates entspricht, auf. Angenommen, die Funktion wäre nicht mit `pub` markiert, könnten wir sie nicht im Integration-Test aufrufen.

Die Verzeichnis- und Dateistruktur sieht damit wie folgt aus:

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

Listing 17–4
Verzeichnisstruktur inklusive Integration-Test

17.3 Ausführung

cargo test

Um alle Tests eines Crates auszuführen, verwenden wir `cargo test`. Als Ergebnis erhalten wir folgende Ausgabe:

Listing 17–5
Ausgabe von cargo test

```
running 1 test
test test_hello_outside_world ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured;
  0 filtered out; finished in 0.00s
```

Ausführung eingrenzen

Um die auszuführenden Tests einzugrenzen, können wir einen Parameter mitgeben: `cargo test el`. Hierbei wird der String `el` in dem Namen der Testfunktion gesucht. Da `el` in `test_hello_outside_world` vorkommt, wird der Test auch ausgeführt.

Ausführung eines dedizierten Tests

Mit dem Ergänzen von `-- --exact` und der Angabe des kompletten Pfades kann auch exakt eine Funktion ausgeführt werden: `cargo test test_hello_outside_world -- --exact`.

Hintergrund

Weitere Möglichkeiten können wir im *Book* unter *Unit Testing* oder in der cargo-Dokumentation unter *cargo test* nachschlagen.

17.3.1 Erwartungen der Testergebnisse (Assertions)

Ein Test wird zu einem Test, indem er eine von uns zuvor formulierte Erwartung prüft. Die Standardbibliothek von Rust bietet zwei Makros an, um erwartete Werte mit den tatsächlichen Ergebnissen zu vergleichen: `assert!`, `assert_eq!`und `assert_ne!`.

Schauen wir uns ein klassisches Beispiel an, in dem wir in einem Test eine bestimmte Zeichenkette erwarten:

```
fn return_string() -> &'static str {
    "Hello"
}

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

    #[test]
    fn pretty_assertions() {
        assert_eq!("Hello", return_string());
    }
}
```

Nachrichten bei fehlgeschlagenen Tests

Für den Fall, dass die Erwartung nicht zutrifft, kann auch eine Nachricht mitgegeben werden, welche Aufschluss über die Erwartung geben kann. Dazu geben wir neben den beiden Werten, die wir vergleichen, eine Zeichenkette mit:

Listing 17–6
assert_eq! mit einer beschreibenden Nachricht

```
assert_eq!("Hello", return_string(), "Hier klappt etwas\
 nicht. Ich erwarte {} - bekomme jedoch {}",
           "Gude", return_string());
```

Wir haben die Zeichenkette der Nachricht mit zwei Platzhaltern versehen, die wir mit den vorne verglichenen Werten versehen. Das ist leider nicht elegant, aber immer noch der Standardweg.

Schönere Assertions

Wir empfehlen als Ergänzung den Einsatz des Crates *pretty_assertions*. Das Crate bringt die oben genannten Makros `assert_eq!` und `assert_ne!` mit und erweitert sie mit einer farblichen Hervorhebung der Unterschiede bei fehlschlagenden Tests. Ein Beispiel können Sie in unserem git-Repository[1] ausprobieren.

17.3.1.1 Erweiterte Assertions mit K9

Ergänzend zum Crate `pretty_assertions`, das die Standard-Makros verschönert, können wir mit dem Crate namens `K9` sogar erweiterte Erwartungen formulieren. Als Beispiel nehmen wir die Prüfung, ob ein `Err` mit einer bestimmten Zeichenkette zurückgegeben wird:

Listing 17–7
Verwendung von k9::assert_err_matches_regex

```
fn return_error() -> Result<&'static str, &'static str> {
    return Err("Das klappt nicht!");
}

#[cfg(test)]
mod tests {
    use super::*;
    use k9::assert_err_matches_regex;

    #[test]
    fn test_error() {
        assert_err_matches_regex!(return_error(), "kl.* nicht");
    }
}
```

Besonders schön sind hierbei die Ausgaben, sobald die Assertion nicht mehr zutrifft. Hier ein Beispiel mit der Prüfung auf den regulären Ausdruck `klappt.*doch`. In der Ausgabe werden die Abweichungen in den Farben grün und rot dargestellt. Da das Buch in Schwarz-Weiß-Druck vorliegt, markieren wir die entsprechenden Stellen in **fett** und legen Ihnen ans Herz, das entsprechende Beispiel[1] in unserem Repository auszuprobieren:

1. *https://www.rust-buch.de/repository/05_testing/src/testing_3_1_1_k9.rs*

Listing 17–8
Verwendung des Makros k9::assert_err_matches_regex als Text

```
assert_err_matches_regex!(return_error(), "klappt doch");

Assertion Failure!

Expected Result<T, E> to be Err(E) that matches
regex when formatted with `format!("{:?}", error)`,

Regex: klappt doch
Formatted error: "Das klappt nicht!"
```

Weitere `assert`-Makros können wir in der *Dokumentation* nachschlagen.

17.3.2 Benennung der Testfunktionen

Die Namen der Funktionen können wir genau wie andere Funktionen in Rust benennen. Theoretisch könnten wir in dem separaten Modul `tests` der Testfunktion den gleichen Namen geben wie der getesteten Funktion. In unserem Unit-Test-Beispiel haben wir die Funktion mit einem Präfix `test_` versehen und damit `test_hello` genannt. Das hat folgende Vorteile:

1. Wir können alle Funktionen aus dem Supermodul mit `use super::*;` im Testmodul zur Verfügung stellen.
2. Wir müssen im Testcode kein Präfix angeben, um eine eventuelle doppelte Benennung auszuschließen.

Dieses Präfix finden wir auch in mehreren Beispielen von Rust-Code, wie zum Beispiel bei *Rust by Example*[2] oder auch beim Framework *Rocket*.

Neben den Spezifika von Rust sollten wir uns darüber hinaus bewusst entscheiden, wie wir die Testfunktionen und -methoden benennen, um den unter Abschnitt 17.1.1 erläuterten Vorteil zu maximieren. Wir wollen bei einem fehlgeschlagenen Test so schnell wie möglich herausfinden, an welcher Stelle unter welcher Bedingung eine Änderung oder ein Fehler passiert ist. Somit können wir den Namen der Funktion dazu nutzen, uns auch Aufschluss über unsere Erwartung zu geben. Gerade bei Tests, die verschiedene Konstellationen oder Logikverzweigungen abdecken, kann uns ein gut gewählter Name enorm helfen. Dazu möchten wir eine Variante ausführlich besprechen.

Einheit_Voraussetzung_Verhalten

Diese Variante setzt die Idee um, indem wir drei Informationen im Namen unterbringen und diese Informationen durch einen Unterstrich (_) trennen. Wir beginnen mit dem Namen der Einheit, die wir testen. Meistens ist das der Name der zu testenden Funktion/Methode. Nehmen

2. *https://doc.rust-lang.org/stable/rust-by-example/*

wir als Beispiel unsere Methode `greet()`. Mit der Idee, die Testfunktion mit `test_` beginnen zu lassen, starten wir also mit `test_greet_`.

Als Zweites definieren wir eine Voraussetzung (oder auch ein Szenario), die wir in unserem Test herstellen. Oft betrifft diese Voraussetzung die Argumente, die wir einer Funktion übergeben. In unserem Beispiel mit greet() könnten wir ein Objekt vom Typ Option übergeben und damit einen Test schreiben, der den Fall von `Option::None` abdeckt.

Als Letztes beschreiben wir das erwartete Verhalten. In unserem Beispiel erwarten wir, anonym begrüßt zu werden, und hätten einen Namen zusammengestellt, der `test_greet_none_greetedanonymously` lautet.

Damit wir uns das im Zusammenhang besser vorstellen können, schauen wir es uns noch mal in einem kompletten Beispiel an. In diesem implementieren wir die eben angedeutete Funktion sowie zwei Tests.

Listing 17–9
Beispiel zur Benennung von Testfunktionen

```
fn greet(greeted: Option<String>) -> String {
    return match greeted {
        Some(greeted) => return format!("Hello {}!", greeted),
        None => "Hello anonymous!".into(),
    };
}

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

    #[test]
    fn test_greet_somename_greetedbyname() {
        assert_eq!(greet(Option::Some(String::from("Marcel"))),
        "Hello Marcel!");
    }

    #[test]
    fn test_greet_none_greetedanonymously() {
        assert_eq!(greet(Option::None), "Hello anonymous!");
    }
}
```

Neben dieser Variante gäbe es noch weitere, wie zum Beispiel das Verhalten in einem Satz (`test_greet_supportsanonymousgreetings`), zu beschreiben. Solange es dem leichten Verständnis dient, können wir es nur unterstützen. Welche Variante wir auch anstreben: Wir sollten stets prüfen, ob es uns hilft. In den Beispielen oben haben wir der Einfachheit halber von diesem Ansatz abgesehen, da es uns nicht geholfen hätte.

17.4 Mocking

Sobald wir uns in der Softwareentwicklung mit Tests auseinandersetzen, lässt das Thema *Mocking* nicht lange auf sich warten. Hierbei tauschen wir im Rahmen von Tests abhängige Codestrukturen, um Logik des zu testenden Codes in jedem Testlauf verlässlich abschreiten zu können.

Weitere Definitionen wollen wir uns an diesem Punkt sparen und verweisen auf den Wikipedia-Artikel *Mock-Objekt*.

Wir wollen uns nun anschauen, wie wir den oben beschriebenen Austausch in Rust bewerkstelligen können.

17.4.1 Erste Schritte ohne Framework

Beim Thema Mocking denken wir Softwareentwickler schnell an die Verwendung einer Bibliothek, wie z. B. *Mockito*, *Powermock* oder *Jest*, die das Mocken vereinfachen soll. Jedoch ist die Einbindung einer solchen nicht immer notwendig. Das gilt auch für die Entwicklung in Java oder JavaScript. Im Folgenden wollen wir uns das autarke Mocken ansehen.

Bei der Verwendung von Java hat es sich etabliert, ein Mock-Objekt mit einem Interface und einer jeweiligen Mock-Implementierung umzusetzen, zum Beispiel:

Listing 17–10
Verwendung von Interfaces in Java

```
interface Greeter {
  public String greet();
}

class GreeterImpl implements Greeter {
  public String greet() {
    return "Hello";
  }
}
```

Ein Trait könnten wir als ein Pendant zum Interface in Java, C# oder PHP auffassen. So liegt es für uns nahe, anzunehmen, dass wir in Rust die Vorgehensweise kopieren können. Leider ist das nicht ganz der Fall.

Bei der Angabe des Typs können wir nicht einfach den Trait angeben. Der Rust-Compiler kann durch eine reine Angabe des Traits nicht die Größe des Objekts bestimmen. Wir können zwar auf Trait-Objekte setzen. Diese sind aber schwerer zu handhaben und haben auch einen (kleinen) Performancenachteil. Einfacher für unser Beispiel ist die Verwendung von generischen Datentypen. Damit wir den Code kompakter anschauen können, teilen wir das Beispiel in zwei Listings auf und beginnen mit den Imports und dem (fiktiven) produktiven Code:

```
pub trait Greeter {
    fn greet(self) -> String;
}

struct GreeterImpl {}

impl Greeter for GreeterImpl {
    fn greet(self) -> String {
        String::from("Hello world!")
    }
}

#[allow(dead_code)]
pub fn use_greeter<G: Greeter>(greeter: G) -> String {
    greeter.greet()
}
```

Listing 17–11
Definition, Implementierung und Verwendung von Greeter

Zunächst definieren einen Trait namens `Greeter`. Dieser enthält die Methode `greet()`, die nichts entgegennimmt, aber eine Zeichenkette, den Gruß, zurückgibt.

Wir definieren einen leeren strukturierten Datentyp namens `GreeterImpl`, um den Trait implementieren zu können.

Eben diese Implementierung erfolgt als Nächstes. Wir implementieren dabei `greet()`, indem wir mit dem allseits bekannten `Hello world!` grüßen.

Dann verwenden wir wie angekündigt den Typ `Greeter` in einem generischen Parameter. Dadurch sind wir gleich in der Lage, jedes Objekt, das den Trait `Greeter` implementiert, zu übergeben. In `use_greeter()` machen wir genau das, was der Funktionsname sagt, und rufen auf dem übergebenen Objekt `greet()` auf. Diese Funktion werden wir gleich durch einen Unit-Test abdecken. Das Attribut `#[allow(dead_code)]` setzen wir, damit der Compiler bei der Übersetzung diese ungenutzte Funktion nicht anstreicht.

Wir machen weiter mit den Tests und damit auch mit der Erstellung des Mocks.

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

    struct MockGreeter {}

    impl Greeter for MockGreeter {
        fn greet(self) -> String {
            String::from("Hello from the mock!")
        }
    }

    #[test]
```

Listing 17–12
Manuelles Mocken von Greeter

```
    fn test_greeter() {
        assert_eq!(use_greeter(MockGreeter{}),
                   "Hello from the mock!");
    }

    #[test]
    fn test_greeter_productive_code() {
        assert_eq!("Hello world!",
                   use_greeter(GreeterImpl {}));
    }
}
```

Wie oben beginnen wir mit der Definition eines separaten Moduls `tests`, markieren es mit dem Attribut `#[cfg(test)]` und importieren alles aus dem Supermodul.

Ähnlich der vorherigen Implementierung im produktiven Code implementieren wir `Greeter` als `MockGreeter`. Allerdings geben wir als Gruß `Hello from the mock!` zurück.

Anschließend testen wir die Funktion `use_greeter()`, indem wir den Mock instanziieren und übergeben. Wir erwarten mit der Verwendung des Markos `assert_eq!`, dass auch der Mock verwendet wird.

Abschließend ergänzen wir noch einen Test, um den produktiven `Greeter` auch noch mal zum Einsatz zu bringen.

17.4.2 Einsatz eines Frameworks: Mockall

In der Rust-Community haben sich mehrere Mocking-Crates hervorgetan. Eine gute Auflistung dieser finden wir unter dem sogenannten *mock_shootout*[3].

Um es kurz zu machen: Der Autor dieser Gegenüberstellung kam zu dem Schluss, dass keines der Frameworks alle Features abdeckt, die er sich gewünscht hat. So schrieb er sein eigenes und nannte es *Mockall*.

Wir können nach Prüfung des Vergleichs sowie aller Details von *Mocktopus* und *Mock-It* den Ausgang des Shootouts bestätigen. Ausschlaggebend dafür war, dass trotz der großen Möglichkeiten, Mocks zu verwenden, Mockall die Übersetzung von produktiven Code gegen die normale Version von Compiler und Bibliotheken (*stable*) unterstützt. Das war aus unserer Sicht mit anderen Crates mit diesem Funktionsumfang nicht möglich.

Im Folgenden wollen wir auf typische Anwendungsfälle eingehen und die Beispiele aus der Doku um komplette und funktionierende Beispiele ergänzen.

---

3. *https://asomers.github.io/mock_shootout/*

17.4.2.1 Traits

Um uns langsam an den Einsatz von Mockall heranzuwagen, schauen wir uns erst einmal das Beispiel an, welches wir zuvor ohne Mockall umgesetzt haben: `Greeter`. Vorab wollen wir festhalten, dass wir die Verwendung von Mockall in vier grobe Schritte aufteilen können:

1. Mockall importieren
2. Trait mit Attribut `#[automock]` markieren
3. Instanz des Mocks erstellen: `Mock[Name des Traits]::new();`
4. Methode des Traits über `expect_[name]()` mit einem Rückgabewert versehen

Schauen wir uns das im Detail an. Zuerst wieder der produktive Code:

```
use mockall::automock;

#[automock]
pub trait Greeter {
    fn greet(self) -> String;
}

struct GreeterImpl {}

impl Greeter for GreeterImpl {
    fn greet(self) -> String {
        String::from("Hello world!")
    }
}

#[allow(dead_code)]
pub fn use_greeter<G: Greeter>(greeter: G) -> String {
    greeter.greet()
}
```

Wir importieren das Makro/das Attribut `automock` und markieren damit den uns bekannten Trait `Greeter`.

Danach geht es mit der Implementierung, wie wir sie bereits von oben kennen, weiter.

Auch hier fahren wir weiter fort mit den Tests.

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

    #[test]
    fn test_greeter() {
        let mut mock = MockGreeter::new();
        mock.expect_greet()
            .return_const("Hello from the mock!");
```