1 Rust – Einführung

In diesem Kapitel werfen wir einen ersten Blick auf Rust-Programme, betrachten die Installation von Rust und der Sprachunterstützung in verschiedenen Entwicklungsumgebungen, sodass wir möglichst schnell praktische Schritte mit der Sprache unternehmen, ein Beispielprogramm schreiben und mit dem Rust-eigenen Build-System übersetzen und starten können.

1.1 Warum Rust?

Rust ist eine moderne Sprache, die sehr stark auf Geschwindigkeit und Parallelverarbeitung ausgelegt ist. Vielfach wird Rust als Systemprogrammiersprache und Ersatz für C dargestellt, der Anwendungsbereich ist aber sehr viel breiter. Betrachten wir ein paar der interessanten Eigenschaften von Rust.

1.1.1 Rust und der Speicher

Das absolute Alleinstellungsmerkmal ist die Art, wie Rust mit Speicher umgeht. Rust kann garantieren, dass durch die Verwaltung des Speichers zur Übersetzungszeit keine Fehler zur Laufzeit auftreten können. Damit braucht Rust auch keinen Garbage Collector. Das verhindert unbeabsichtigte Unterbrechungen im Programmablauf, um den Speicher aufzuräumen. Wir haben also nicht nur korrektere Programme, die schneller laufen, sie verhalten sich auch deterministischer.

Um dies zu erreichen, wird für jeden Wert ein Eigentümer festgelegt. Dies kann ein primitiver Wert sein oder eine beliebig komplexe Struktur. Ein Wert lebt, solange der Eigentümer lebt.

Der Eigentümer kann wechseln, und für den Zugriff auf ein Objekt können Referenzen ausgeliehen werden (*Borrowing*). Ausgeliehene Referenzen sind im Normalfall Lesereferenzen, es kann aber alternativ auch maximal eine Schreib-/Lese-Referenz auf einen Wert

definiert werden. Dies impliziert, dass wir keine aktive Lesereferenz haben. Die Beschränkung auf eine einzige schreibende Instanz sorgt bei Neulingen meist für Überraschungen, hat aber den großen Vorteil, dass es keine undefinierten Zustände durch gleichzeitiges Schreiben oder nicht synchronisiertes Lesen geben kann.

Dieses *Ownership* genannte Konzept ist extrem mächtig, braucht aber zum vollständigen Verinnerlichen etwas Zeit und Übung. Wir werden dies in Abschnitt 7.2 kennenlernen und in Kapitel 15 im Detail beleuchten.

1.1.2 Rust und Objektorientierung

Rust ist eine Programmiersprache, die mit der Kapselung von Daten und Funktionen und Methoden auf diesen Daten objektorientierte Konzepte unterstützt.

Rust erreicht dies durch die Einführung von Modulen, die private und öffentliche Daten und Funktionen enthalten. Polymorphismus wird durch das Konzept der *Traits* erreicht, die inzwischen in vielen anderen Programmiersprachen wie Kotlin oder Scala auch verwendet werden. Eine vergleichbare Funktionalität gibt es in Java seit der Version 8 mit den Default-Methoden in Interface-Spezifikationen.

Rust bietet allerdings anders als die gewohnten objektorientierten Sprachen keine Vererbung. Dies mag im ersten Moment überraschen und ist eine Abkehr vom normalen objektorientierten Denken, hat aber gute Gründe.

Aus konzeptioneller Sicht ist es problematisch, dass wir bei der Vererbung nicht kontrollieren können, welche Teile unserer Elternklasse wir erben möchten. Dies kann dazu führen, dass wir in abgeleiteten Klassen Funktionalität haben, die dort nicht gewollt ist.

Das praktischere Argument ist aber, dass durch Verzicht auf Vererbung ein hoher Aufwand zur Identifikation der richtigen auszuführenden Methode/Funktion wegfällt. Dies macht Rust-Programme deutlich laufzeiteffizienter.

Wir werden uns mit objektorientierten Konzepten in Kapitel 9 auseinandersetzen.

1.1.3 Rust und funktionale Programmierung

Zur Unterstützung funktionaler Programmierung bietet Rust *Closures*, anonyme Funktionen, die auf ihre Umgebung zur Zeit der Definition zugreifen können. Dieses vielseitige Konstrukt findet sich in mehr und mehr Sprachen und erlaubt eine sehr elegante Kapselung von Funktionalität und Daten.

Zusammen mit Iteratoren, die die Verarbeitung von Sammlungen von Daten kapseln, erlauben Closures sehr mächtige funktionale Abstraktionen. Iteratoren und Closures werden wir in Kapitel 11 kennenlernen.

1.1.4 Rust und Parallelverarbeitung

Rust bietet eine direkte Abstraktion der Thread-Funktionalität des unterliegenden Betriebssystems. Dies sorgt für den geringstmöglichen Mehraufwand zur Laufzeit, beschränkt aber natürlich die Flexibilität in der Verwendung von Threads auf die Unterstützung durch das unterliegende System. Bei Bedarf können allerdings auch Thread-Module verwendet werden, die eine unabhängige und damit flexiblere Implementierung anbieten. Dies erlaubt uns, von Fall zu Fall zu entscheiden, ob wir die größere Flexibilität oder den geringeren Speicherbedarf bevorzugen. Während die Entscheidung in vielen Fällen in Richtung der Flexibilität getroffen werden wird, gibt es eingeschränkte Umgebungen (wie zum Beispiel Mikro-Controller), in denen die Möglichkeit der expliziten Wahl sehr vorteilhaft ist.

Viele der Probleme, die bei der normalen Programmierung von paralleler Verarbeitung zu sehr hoher Komplexität und damit zu schwer auffindbaren Fehlern führen, finden wir in Rust nicht. Dies entsteht durch das *Ownership*-Modell, das dafür sorgt, dass der Compiler problematische Stellen im Quelltext sehr früh identifizieren und damit entfernen kann. Das heißt nicht, dass Rust alle Probleme im Zusammenhang mit Parallelprogrammierung löst. Es erlaubt uns aber, uns auf die wirklich schwierigen Probleme zu konzentrieren.

Threads in Rust können kommunizieren, indem sie Nachrichten in verschiedene Kanälen senden oder aus diesen empfangen. Zusätzlich können sie Teile ihres Zustands geschützt durch eine Mutex-Abstraktion mit anderen Threads teilen.

Parallelprogrammierung in Rust ist sehr mächtig, und wir werden uns in Kapitel 16 eingehend damit beschäftigen.

1.2 Ein Beispielprogramm

Als ein erstes Beispiel, um Ihren Appetit für Rust zu wecken, betrachten wir ein kleines Programm, das bereits viele Eigenschaften von Rust zeigt. Da dieses deutlich über das klassische »Hallo Welt«-Programm hinausgeht, sollten Sie sich keine Sorgen machen, wenn einzelne Funktionalitäten noch nicht vollständig klar sind. Die Erklärungen sind an dieser Stelle notwendigerweise etwas kurz, alle angesprochenen Eigenschaften werden wir später deutlich detaillierter betrachten.

Listing 1–1
Zeilenweises Lesen und Ausgabe einer Datei

```
use std::fs::File as Datei;
use std::io::{BufReader, BufRead};

fn main() {
    let file = Datei::open("hallo.txt")
        .expect("Konnte Datei nicht öffnen");
    let reader = BufReader::new(file);

    for line in reader.lines() {
        let line = line
            .expect("Konnte Zeile nicht lesen");
        println!("{}", line);
    }
}
```

Wir beginnen mit dem Import benötigter Funktionalität (wie auch aus Java bekannt). Wir benennen den aus dem Namensraum beziehungsweise Modul `std::fs` (durch den Pfadtrenner `::` getrennte Namen sind hierarchische Pfadangaben) importierten Typ `File` um in `Datei` und importieren im nächsten Schritt die beiden Typen `BufReader` und `BufRead`. Der Typ `BufReader` unterstützt gepuffertes Lesen aus einer Quelle und ist damit deutlich effizienter als ein direktes Lesen.

Dann folgt die Definition unserer ersten Funktion, gekennzeichnet durch das Schlüsselwort `fn`. In unserem Fall ist dies die Funktion `main()`, die keine Parameter und keinen Rückgabewert hat. Der Körper der Funktion findet sich im durch geschweifte Klammern definierten Block von Anweisungen, die durch Semikolon getrennt sind. Wie in vielen anderen Sprachen hat diese Funktion eine Sonderrolle: Sie ist der Einstiegspunkt in unser Programm und wird als Erstes aufgerufen, um den Programmabfluss zu starten.

Wir versuchen mit der Methode `open()` (eine Methode des Typs `Datei`, unserem umbenannten Typ `File`) eine Datei mit dem Namen *hallo.txt* zu öffnen. Dieser Aufruf liefert ein Objekt vom Typ `Result` zurück, das entweder das Ergebnis des erfolgreichen Aufrufs oder den durch den Aufruf ausgelösten Fehler enthält. Die Funktion `expect()` nimmt dieses Objekt und liefert im Erfolgsfall das Ergebnis zurück, im Fehlerfall wird die als Parameter übergebene Nachricht ausgegeben und das Programm mit einem Fehler beendet.

Hintergrund

Tatsächlich erzeugt die Funktion `expect()` einen nichtbehandelbaren Fehler vom Typ `std::Panic`. Dies ist ein völlig normaler Weg für Rust, Probleme zu behandeln, solange es der eigene Quelltext ist oder wir uns in der Prototypphase befinden. Im Falle von Produktionssoftware oder aber Bibliotheken sind andere Wege zur Fehlerbehandlung zu bevorzugen.

Wir weisen das Ergebnis, ein Objekt vom Typ `Datei`, der Variable `file` zu und erzeugen im nächsten Schritt ein Objekt vom Typ `BufReader` darauf, das wir in der Variable `reader` halten.

Nun iterieren wir mit einer `For`-Schleife über die einzelnen Zeilen der Datei in einem Iterator, den wir über den Aufruf von `reader.lines()` erhalten. Hierbei ist wichtig, dass die Funktion `lines()` die Zeilen ohne abschließenden Zeilenvorschub liefert. Der nachfolgende Block (durch geschweifte Klammern definiert) wird für jede Zeile ausgeführt. Das Ergebnis des Leseversuches landet als `Result`-Objekt in der Variablen `line`.

Auch hier extrahieren wir die eigentliche Zeichenkette wieder mit einem Aufruf der Funktion `expect()` mit einer Fehlermeldung, falls das Lesen nicht erfolgreich war. Das Ergebnis weisen wir der Variablen `line` zu. Der Effekt ist, dass wir keinen Zugriff mehr auf das `Result`-Objekt haben, das uns der Iterator zurückgegeben hat, sondern jetzt das eigentliche Resultat, die Zeichenkette mit der aktuellen Zeile der Datei, verwenden.

Hintergrund

Diese *Shadowing* genannte Funktionalität von Rust ist in vielen Fällen sehr vorteilhaft und elegant, kann aber bei Missbrauch zu schlechter Lesbarkeit führen. Der Umgang mit `Result`-Objekten ist einer der Fälle, in denen dies die Absicht des Programmierers klar kommuniziert.

Die letzte Anweisung unseres Programmes ist der Aufruf des Makros `println!`, das die Argumente mit einem abschließenden Zeilenvorschub ausgibt, so wie es die Formatzeichenkette (das erste Argument) vorgibt. Die Variante ohne Zeilenvorschub heißt `print!`.

Makros werden gekennzeichnet durch das Ausrufezeichen am Ende des Namens. Makros haben insbesondere aufgrund der Herausforderungen im Zusammenhang mit dem C-Präprozessor einen schlechten Ruf. In C entsteht dieser aus der direkten Ersetzung von Makroaufrufen durch ihren Inhalt, ohne dass der Präprozessor in irgendeiner Weise prüft, ob die Änderung syntaktisch korrekten Quelltext hinterlässt. Rust-Makros sind hier deutlich besser, da die Umsetzung durch den Compiler erfolgt und grundsätzlich gültige Ausdrücke erzeugt werden.

Das Makro `println!` ist ein exzellentes Beispiel für die Eleganz von Makros. In Rust müssen wir Funktionen mit der vollständigen Anzahl ihrer Parameter definieren, ein wichtiger Teil der Funktionalität von `println!` ist aber gerade, mit einer beliebigen Zahl von Parametern

Hintergrund

Es gibt zwei Arten von Makros in Rust, die deklarativen und die prozeduralen. In beiden Fällen übernimmt der Rust-Compiler die Aufgabe der Makroübersetzung, was Fehler sehr viel schneller und besser erkennen lässt.

Die deklarativen Makros sind relativ einfach zu schreiben, aber in ihrer Mächtigkeit etwas beschränkt. Wir werden später ein eigenes definieren.

Prozedurale Makros in Rust sind eine elegante Art der Metaprogrammierung, anders als der C-Präprozessor, der nur einfache Textersetzungen durchführt. Sie operieren direkt auf dem *Abstract Syntax Tree,* den der Rust-Compiler aus dem Quelltext erzeugt. Dies erlaubt eine extrem hohe Mächtigkeit dieser Art von Makros, dafür sind sie schwerer zu schreiben.

umgehen zu können. Das Rust-Makro `println!` erzeugt nun aus dem jeweiligen Quelltext den korrekten Aufruf der zugehörigen Bibliotheksfunktionen. Dies führt dazu, dass wir `println!` mit einer der Formatierungszeichenkette entsprechenden Anzahl von Argumenten aufrufen können.

Das erste Argument von `println!` ist diese Formatierungszeichenkette (ein Literal), die Formatierungsanweisungen und Platzhalter für die Ausgabe enthält. Die Zeichenkette muss ein Literal sein, da das Makro aus dieser den eigentlichen Code generiert (präziser: die Manipulationen des *Abstract Syntax Tree* durchführt).

In unserem Fall enthält diese Zeichenkette einfach einen Platzhalter `{}`, der durch das zweite Argument, unsere aktuelle Zeile, belegt wird. Dies führt dazu, dass alle Zeilen der Datei *hallo.txt* ausgegeben werden.

1.3 Installation von Rust

Die Rust-Entwicklung schreitet sehr schnell voran. Um dies zu reflektieren, werden in vergleichsweise kurzen Abständen (zum Zeitpunkt der Veröffentlichung alle 6 Wochen) neue Versionen von Rust veröffentlicht. Um die Installation jederzeit aktuell halten zu können, einfach zwischen verschiedenen Kanälen (*stable, beta, nightly*) wechseln zu können oder zum Beispiel Übersetzungen für andere Zielarchitekturen zu ermöglichen, stellt das Rust-Projekt das Werkzeug `rustup` zur Verfügung, das die Installation und Aktualisierung sehr stark vereinfacht. Hierbei werden normalerweise alle Werkzeuge im Verzeichnis `.cargo` im Benutzerverzeichnis installiert. Konfigurationsoptionen erlauben aber auch eine systemweite Installation.

Natürlich stehen auch jeweils aktuelle Installationspakete für die manuelle Installation bereit, aber für die Entwicklung mit Rust ist die Verwendung von `rustup` die beste Wahl.

1.3.1 Installation von rustup

Detaillierte Anweisungen inklusive aller Varianten zur Installation von `rustup` finden sich unter:

https://rust-lang.github.io/rustup/installation/index.html

Deshalb werden wir hier nur den einfachen Installationspfad betrachten.

1.3.1.1 Windows

Gehen Sie zur Website

https://www.rust-lang.org/tools/install

und laden Sie `Rustup-Init.exe` herunter. Nach der Ausführung des Installationsprogrammes können Sie den Erfolg der Installation testen, indem Sie ein CMD-Fenster öffnen und `rustc –version` eingeben. Falls dies nicht klappt, prüfen Sie, ob der Pfad korrekt erweitert wurde, und versuchen Sie den Aufruf mit `%userprofile%/.cargo/bin/rustc –version`.

1.3.1.2 Andere Systeme

Die allgemeine Methode, um `rustup` zu installieren, funktioniert für OSX, Linux, aber auch für das Linux-Subsystem unter Windows. Hierbei wird ein Skript ausgeführt, das von einem Server heruntergeladen wird. Man kann argumentieren, dass dies gefährlich ist. Es besteht aber natürlich die Möglichkeit, das Skript vor der Ausführung zu betrachten und zu prüfen. Es prüft, auf welcher Plattform es läuft, wählt dementsprechend das Installationsprogramm aus, lädt es herunter und führt es aus. Führen Sie das Skript mit dem folgenden Befehl aus:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Wenn Sie dem Skript nicht trauen, dann können Sie das Installationsprogramm auch von Hand identifizieren und herunterladen.

1.3.1.3 Installation über Paketmanager

Neben der direkten Installation gibt es auch die Möglichkeit der Installation über Paketmanager. Unter OSX gibt es `Homebrew` und `MacPorts` als bekannteste Paketmanager, unter Windows gibt es `Chocolatey` oder `Scoop`. Auch unter gängigen Linux-Distributionen gibt es die Möglich-

keit der Installation über Paketmanager, für die Entwicklung wird aber generell die Verwendung von `rustup` empfohlen.

Auch hier besteht natürlich das Problem des Vertrauens, es kann aber durchaus sinnvoll sein, dem Ersteller eines Pakets für den verwendeten Paketmanager mehr zu vertrauen als einem Skript von einem Webserver. Diese Entscheidung liegt alleine bei Ihnen.

1.4 IDE-Integration

Neben der direkten Verwendung der durch `rustup` zur Verfügung gestellten Werkzeuge auf der Kommandozeile gibt es auch sehr gute Unterstützung für die Programmierung in Rust durch verschiedene Entwicklungsumgebungen (IDE – Integrated Development Environment). Insbesondere das von den IDEs angebotene Auto-Vervollständigen, die Auflistung der Parameter von aufgerufenen Funktionen und die Unterstützung für Refactoring machen die Entwicklung deutlich effizienter. Die starke Integration eines Debuggers und einer Versionsverwaltung tut ihr Übriges, um schnell Software zu entwickeln.

1.4.1 Rust Language Server und Rust-Analyzer

Rust bietet zur Unterstützung von Entwicklungsumgebungen seit langer Zeit bereits den Rust Language Server RLS an, der im Hintergrund läuft und die IDE durch Informationen zu verwendeten Symbolen unterstützt. Diese Unterstützung beinhaltet Dokumentation, Umformatierung, Autovervollständigung, Refactoring, das Auffinden der Definition eines Symbols (dies ermöglicht in der Entwicklungsumgebung das Springen zur Funktion, die man aufruft) oder auch die Übersetzung im Hintergrund. Dies funktioniert in den meisten Fällen auch akzeptabel, allerdings wird RLS seit längerer Zeit nicht mehr weiterentwickelt und ist im Wartungsmodus.

Hintergrund

Die Idee eines Language Servers und des damit verbundenen Protokolls LSP (Language Server Protocol) wurde ursprünglich von Microsoft für Visual Studio Code entwickelt. Das dahinterliegende Konzept ist, dass der Aufwand für die Entwicklung von sprachspezifischen Funktionen wie Syntaxhervorhebung, Autovervollständigung, Refactoring bis hin zur Übersetzung aus der Entwicklungsumgebung extrahiert und in einen eigenen Prozess ausgelagert wird. Dies erlaubt die Entkopplung und Verwendung des Language Servers in verschiedenen Entwicklungsumgebungen. Das zugehörige Protokoll wurde standardisiert und wird mehr und mehr auch von anderen Entwicklungsumgebungen verwendet.

Die Alternative ist der Rust-Analyzer, eine Neuimplementierung des RLS. Dieser ist zwar noch in einer frühen Phase, trotzdem aber schon weiter entwickelt als RLS und bietet eine deutlich vollständigere Unterstützung. Nachdem inzwischen auch das Rust-Projekt selbst an einer Transition von RLS zu Rust-Analyzer arbeitet und sogar der Originalentwickler des RLS ganz explizit sagt, man solle Rust-Analyzer verwenden, empfehlen auch wir die Verwendung des Rust-Analyzers anstelle des RLS. Dieser wird zwar (zum Zeitpunkt der Veröffentlichung) immer noch als Preview und Alphaversion bezeichnet, wir haben aber mit dieser Implementierung nur positive Erfahrungen gemacht.

1.4.2 Visual Studio Code

Visual Studio Code ist eine kostenlose IDE von Microsoft, die auf dem Electron-Framework basiert. Damit ist sie plattformübergreifend in allen gängigen Systemumgebungen verfügbar. Visual Studio Code bietet zwei Erweiterungen, die die Entwicklung mit Rust unterstützen. Sie können diese IDE hier herunterladen:

https://code.visualstudio.com/

1.4.2.1 Erweiterungen zur Entwicklung mit Rust

Es gibt zwei Erweiterungen, die die Entwicklung von Rust in Visual Studio Code unterstützen: »Rust for Visual Studio Code« und »Rust-Analyzer«. Die erste Erweiterung nutzt den Rust Language Server, die zweite den Rust-Analyzer.

Da der Rust-Analyzer der von Rust empfohlene Language Server ist, ist die zugehörige Erweiterung die logische Wahl. Auch diese Erweiterung wird wie der Rust-Analyzer selbst (zum Zeitpunkt der

Veröffentlichung) immer noch als Preview und Alphaversion bezeichnet, unsere Erfahrungen sind aber ausnahmslos positiv.

Zur Installation wechseln Sie in die Extensions-Sicht und geben in dem Suchfeld »Rust« ein. Die beiden beschriebenen Plugins sollten direkt angezeigt werden. Wählen Sie das »Rust-Analyzer«-Plugin aus und klicken Sie auf »Install«. Das Plugin installiert den Rust-Analyzer mit, sodass sie hier keinen zusätzlichen Aufwand haben.

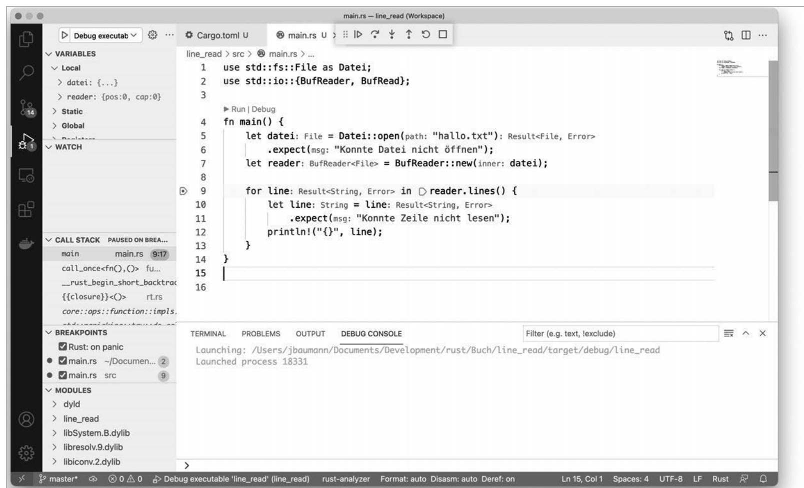


Abb. 1–1
Debugging-Session mit Visual Studio Code

1.4.3 IntelliJ IDEA

IntelliJ IDEA ist eine sehr leistungsfähige Entwicklungsumgebung der Firma JetBrains, die es sowohl in einer kostenlosen Community-Version als auch in einer kommerziellen Ultimate-Version gibt. Sie finden die verschiedenen Versionen der IDE hier:

https://www.jetbrains.com/de-de/idea/

Das für diese IDE verfügbare Rust-Plugin ist unabhängig sowohl von RLS als auch von Rust-Analyzer implementiert, und es bietet eine sehr weitreichende Unterstützung für Rust an. Insbesondere die Refactoring-Unterstützung ist vorbildlich.

Der Nachteil ist allerdings, dass Debugging mit dem Rust-Plugin nur in der Ultimate-Edition freigeschaltet wird, es also keine kostenlose Unterstützung für Debugging gibt.

Aber auch ohne Debugging bietet das Plugin eine große Menge an Funktionalität und ist empfehlenswert, insbesondere wenn IntelliJ bereits für andere Projekte in Verwendung ist.

Zur Installation wechseln Sie in die Einstellungen (*Preferences*), wählen dort Plugins, suchen nach »Rust« und installieren das Rust-Plugin.

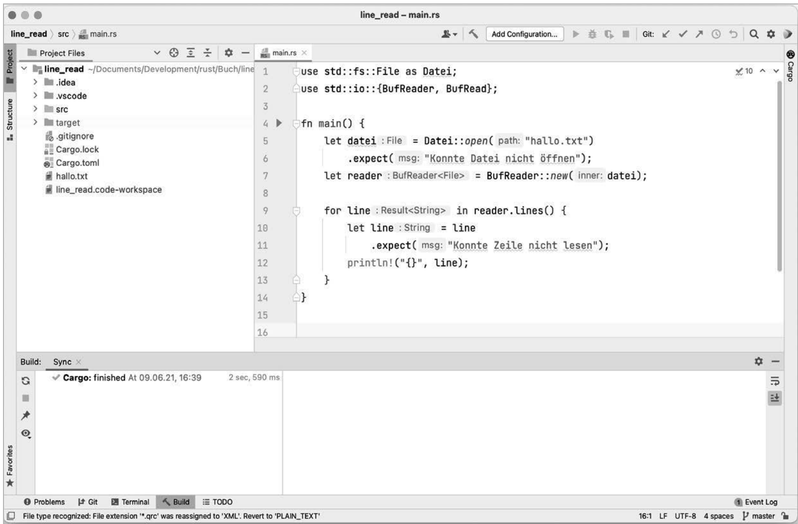


Abb. 1–2
Jetbrain Intellij IDEA in Aktion

1.4.4 Eclipse

Eclipse bietet mit der »Corrosion«-Erweiterung eine Unterstützung für Rust-Programmierung, die auf dem Rust-Analyzer basiert.

Diese Erweiterung für Eclipse ist gefühlt noch nicht ganz so weit wie zum Beispiel die Unterstützung in Visual Studio Code, aber durchaus verwendbar. Aufgrund der Tatsache, dass der Rust-Analyzer verwendet wird, stehen sämtliche der hierdurch bereitgestellten Funktionalitäten ähnlich zur Verfügung wie in Visual Studio Code. Sie finden die verschiedenen Versionen der Eclipse-IDE hier:

https://www.eclipse.org/downloads/

Corrosion erwartet eine Installation des Rust-Analyzers, die Sie getrennt vornehmen müssen. Das jeweils aktuelle Release finden Sie auf Github zum Herunterladen.

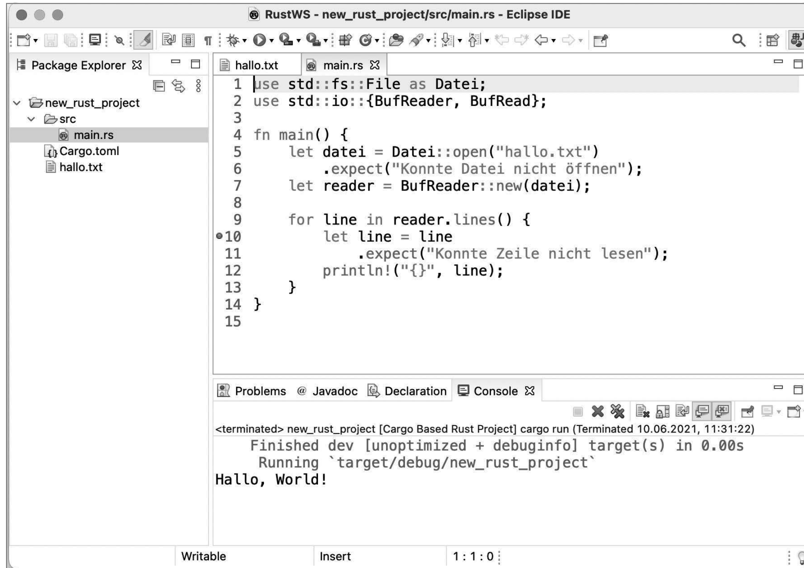


Abb. 1–3
Programmausführung mit Eclipse Corrosion

1.4.5 Welche Entwicklungsumgebung ist die beste?

Aufgrund der sehr guten Unterstützung der Sprachspezifika durch den Rust-Analyzer, der sowohl von Visual Studio Code als auch von Eclipse Corrosion integriert wird, lässt sich die Entscheidung frei nach dem eigenen Geschmack treffen, wenn Sie eine kostenlose Entwicklungsumgebung nutzen wollen. Egal ob Sie sich mit Eclipse oder mit Visual Studio Code wohler fühlen, Sie bekommen in beiden Fällen eine gute Unterstützung.

Wenn Sie bereit sind, Geld auszugeben, oder falls Sie die Ultimate Edition von IntelliJ IDEA für andere Zwecke bereits erworben haben, dann haben Sie hier eine fantastische Unterstützung durch das zugehörige Rust-Plugin.

Zu guter Letzt können Sie auch mit einem Editor wie dem VIM oder Emacs mit Syntaxhervorhebungen gut arbeiten. Diese bieten ebenso Unterstützung für das Language-Server-Protokoll an und damit ähnliche Funktionalität wie die bereits genannten Entwicklungsumgebungen.

Werkzeuge

Wenn wir uns über die Kommandozeile Gedanken machen, kommen wir irgendwann auch zum Thema Debugging. Rust bietet nicht nur die Unterstützung für den seit 30 Jahren konstant weiterentwickelten GDB an, sondern auch den neueren LLDB, der auf der LLVM-Infrastruktur basiert (auch Visual Studio Code bietet die Möglichkeit, nicht nur GDB zu verwenden, sondern über eine Erweiterung auch den LLDB). Die zugehörigen Kommandos lauten `rust-gdb` und `rust-lldb`.

Aktuell schlagen wir die Verwendung von GDB vor, da es für diesen eine schier endlose Menge an Frontends gibt, die die Verwendung vereinfachen.

1.5 Unsere erste praktische Erfahrung

Nachdem wir jetzt einen ersten Blick auf die Rust-Syntax geworfen, Rust auf unserem System installiert und uns die zur Verfügung stehenden Entwicklungsumgebungen kurz angeschaut haben, wollen wir die ersten praktischen Schritte machen.

Für diese Erfahrung wählen wir das klassische HelloWorld-Programm, mit dem wir die ersten Tests der Rust-Werkzeuge durchführen. Im folgenden Listing finden wir den Quelltext für dieses simple Programm. Natürlich können Sie genauso gut das Programm aus unserem ersten Listing verwenden.

Listing 1–2
Das klassische Hello-World-Programm

```
fn main() {
    println!("Hallo Welt!");
}
```

Wir definieren die Funktion `main()`, in der wir das Makro `println!` aufrufen mit der Zeichenkette »Hallo Welt!«. Wir speichern dieses Programm unter dem Namen `hallo_welt.rs` (`.rs` ist die Endung, die typischerweise für Quelltext in Rust verwendet wird).

Aufruf des Rust-Compilers

Um dieses Programm zu übersetzen, rufen wir den Rust-Compiler auf der Kommandozeile auf:

```
> rustc hallo_welt.rs
```

Der Compiler übersetzt jetzt den Quelltext und produziert ein ausführbares Programm mit dem gleichen Namen wie der Quelltext `hallo_welt`. Wir starten das Programm auf der Kommandozeile:

```
> ./hallo_welt
Hallo Welt!
>
```

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

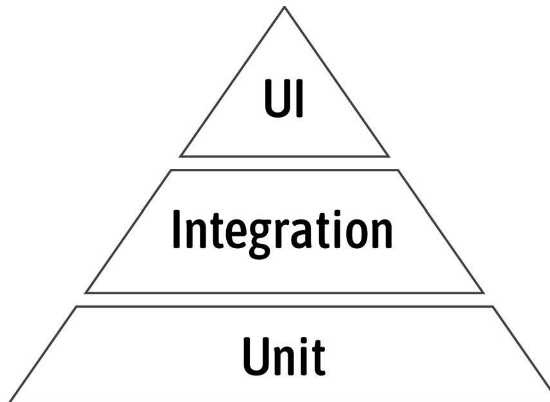
Bevor wir uns die Entwicklung konkreter Tests in Rust anschauen, wollen wir versuchen, eine Definition und Abgrenzung zu finden.

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:

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

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

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:

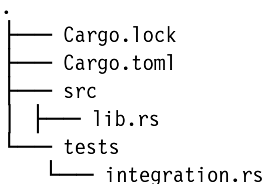
Listing 17–3
Integration-Test von hello_outside_world()

```
#[test]
fn test_hello_outside_world() {
    assert_eq!(rustbuch_testing::hello_outside_world(),
               "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:

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());
    }
}
```

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:

Nachrichten bei fehlgeschlagenen Tests

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

Listing 17–6
assert_eq! mit einer beschreibenden Nachricht

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:

```
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");
    }
}
```

Listing 17–7
Verwendung von k9::assert_err_matches_regex

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:

Listing 17–11
Definition, Implementierung und Verwendung von Greeter

```
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()
}
```

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.

Listing 17–12
Manuelles Mocken von Greeter

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

    struct MockGreeter {}

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

    #[test]
```

```
    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()
}
```

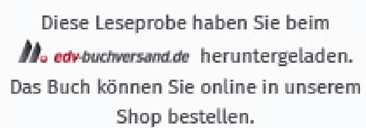
Wir importieren das Makro/das Attribut `automock` und markieren damit den uns bekannten Trait `Greeter`.

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!");
```


Diese Leseprobe haben Sie beim edv-buchversand.de heruntergeladen. Das Buch können Sie online in unserem Shop bestellen.

Hier zum Shop