Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Felix Runge

Die Programmiersprache Rust - Einsatz und moderne Entwurfsmuster in der systemnahen Programmierung dargestellt und auf einer Plattform zentralisiert

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Felix Runge

Die Programmiersprache Rust - Einsatz und moderne Entwurfsmuster in der systemnahen Programmierung dargestellt und auf einer Plattform zentralisiert

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stephan Pareigis
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 14. April 2016

Felix Runge

Thema der Arbeit
Die Programmiersprache Rust - Einsatz und moderne Entwurfsmuster in der systemnahen Programmierung dargestellt und auf einer Plattform zentralisiert

Stichworte
Rust, systemnahe Programmierung, Moderne Entwurfsmuster, Endlicher Automat

Kurzzusammenfassung
Die systemnahe Programmierung gewinnt aufgrund vieler Themen, wie zum Beispiel dem Internet of Things, immer mehr an Bedeutung. Sprachen wie C/C++ bilden Standards in diesem Bereich, aber die junge Sprache Rust versucht durch moderne Konzepte und Prinzipien, wie dem Ownership Modell, eine Alternative darzustellen. In dieser Arbeit wird die Sprache Rust untersucht und etablierte Konzepte und moderne Entwurfsmuster in Rust implementiert. Dabei werden zum einen das Builder-, das Abstract Factory- und das Factory-Pattern, und darüber hinaus ein deterministischer endlicher Automat umgesetzt und die Umsetzung analysiert und bewertet. Dies soll den Anstoß für eine zentrale Plattform zur Sammlung von Problemlösungen in der systemnahen Programmierung darstellen.
Felix Runge

Title of the paper
The Rust Programming Language - Use and Modern Design Patterns in Systems Programming illustrated and centralised on a Platform

Keywords
Rust, systems programming, modern design patterns, finite-state machine

Abstract
Due to many topics like the Internet of Things, systems programming is getting more and more important. Languages like C/C++ are mostly used but the young language Rust tries to be an alternative through its principles and concepts like the Ownership system. This thesis examines the Rust programming language, and established concepts and modern design patterns will be implemented. The builder, abstract factory and factory pattern, and a finite-state machine will be implemented and evaluated. Thus a centralised platform for problems in systems programming should be advanced.

Inhaltsverzeichnis

Listings

1 Einleitung

Die systemnahe Programmierung spielt eine immer wichtigere Rolle in unserer heutigen Welt. Ein großer Teil davon findet sich in eingebetteten Systemen wieder, die auch aufgrund von Themen wie dem *Internet of Things* immer mehr an Bedeutung gewinnen[8]. Da diese Systeme selbstständig arbeiten, und dabei stabil über lange Zeiträume agieren sollen, wird das Thema der ausfallsicheren Programmierung immer wichtiger. Über viele Jahre haben sich unter anderem Entwurfsmuster etabliert, die Ansätze zur Lösung vieler bekannter Probleme, wie zum Beispiel der Erzeugung von Objekten (Erzeugungsmuster) bieten. Durch die Wiederverwendbarkeit [7, S. 1] von Software kann eine schnellere Entwicklung von Softwareprojekten, zum Beispiel im Bereich des Internet of Things, stattfinden. Das Buch der Gang of Four (nachfolgend GoF, [7]), *Design Patterns. Elements of Reusable Object-Oriented Software*, stellt dabei eines Standardwerke für die Programmiermethodik dar.

Zwar verfügt man heute durch das Internet über eine sehr große Anzahl an Ressourcen zur Programmierung, allerdings sind die Informationen meistens auf mehreren Webseiten verteilt. Abhilfe könnte dabei eine zentrale Plattform schaffen, die Hilfe zur Problemlösung von häufig auftretenden Problemen in der systemnahen Programmierung bietet und dazu beispielhafte Implementation von modernen Entwurfsmustern in verschiedenen Programmiersprachen anbietet.

Diese Arbeit beschäftigt sich hauptsächlich mit der Programmiersprache Rust[1]. Rust ist eine vergleichsweise junge, systemnahe Programmiersprache, die mit seinen Konzepten und Prinzipien, wie z.B. dem Ownership-Modell[24], das ausfallsichere Programmieren erleichtern soll. Die Sprache hat dabei das Ziel so schnell wie etablierte Sprachen wie C/C++ zu sein und ist dies bereits[22]. Das bisher größte Projekt, in dem Rust produktiv eingesetzt wird, stellt Servo[2] dar[22]. Servo ist die parallelisierte Browser Engine, die von Mozilla entwickelt und von Samsung unterstützt wird. Darüber hinaus wird Rust bereits in vielen anderen Firmen

[1] https://www.rust-lang.org/
[2] https://servo.org/

und Projekten produktiv eingesetzt. Darunter befinden sich Dropbox [16], Skylight[3] der Firma Tilde[11] und auch das SAFE Network[4] von Maidsafe[13].

1.1 Zielsetzung der Arbeit

Das Ziel der Arbeit stellt die exemplarische Umsetzung von Elementen, die in der systemnahen Programmierung Gebrauch finden, in Rust dar. Dabei werden Erzeugungsmuster (Builder-, Abstract Factory und Factory-Pattern) implementiert und die Implementation erläutert und analysiert. Darüber hinaus wird ein deterministischer endlicher Automat implementiert. Die Implementationen werden dabei an bekannte Ansätzen (z.B. nach GoF) angelehnt und darüber hinaus an die Programmiersprache angepasst, um die Stärken von Rust darzustellen. Um die Problemlösungen umzusetzen, gibt es am Anfang zunächst eine Einführung in die Programmiersprache Rust, die die wichtigsten Elemente der Sprache erklärt. Diese Einführung ist an das offizielle Buch über Rust[23][5] angelehnt und nicht allumfassend.

1.2 Abgrenzung

Diese Arbeit hat nicht den Anspruch, moderne Entwurfsmuster und andere Elemente der systemnahen Programmierung zu analysieren oder diskutieren, sondern nur die Implementation dieser. Des Weiteren soll die Einführung in Rust eine Grundlage für die Problemlösung legen und keine vollständige Erklärung der Sprache darstellen. Darüber hinaus ist die eigentliche Entwicklung einer Plattform nicht Thema dieser Arbeit.

[3] https://www.skylight.io/, Profiler für Ruby on Rails Applikationen
[4] http://maidsafe.net/, Projekt zum Erstellen eines dezentralen Internets
[5] Das Buch ist während der Erstellung der Arbeit nur online verfügbar und wird daher über Kapitel referenziert

2 Die Plattform

Das weiterführende Ziel, zu dem auch diese Arbeit beitragen soll, ist die Erstellung einer öffentlichen Plattform zur Lösung von Problemen im Bereich der Programmierung mit reaktiven Systemen. Zunächst soll die Plattform von Studenten und unter der Leitung von Herrn Prof. Dr. Stephan Pareigis betrieben werden. Diese Plattform soll einen problemorientierten Ansatz verfolgen. Das heißt, man sucht anhand eines Problems und findet passende und bereits etablierte Lösungsvorschläge zur Lösung des Problems.[1]

Dazu gibt es schon einige definierte Kategorien, die im folgenden Bild zu sehen sind.

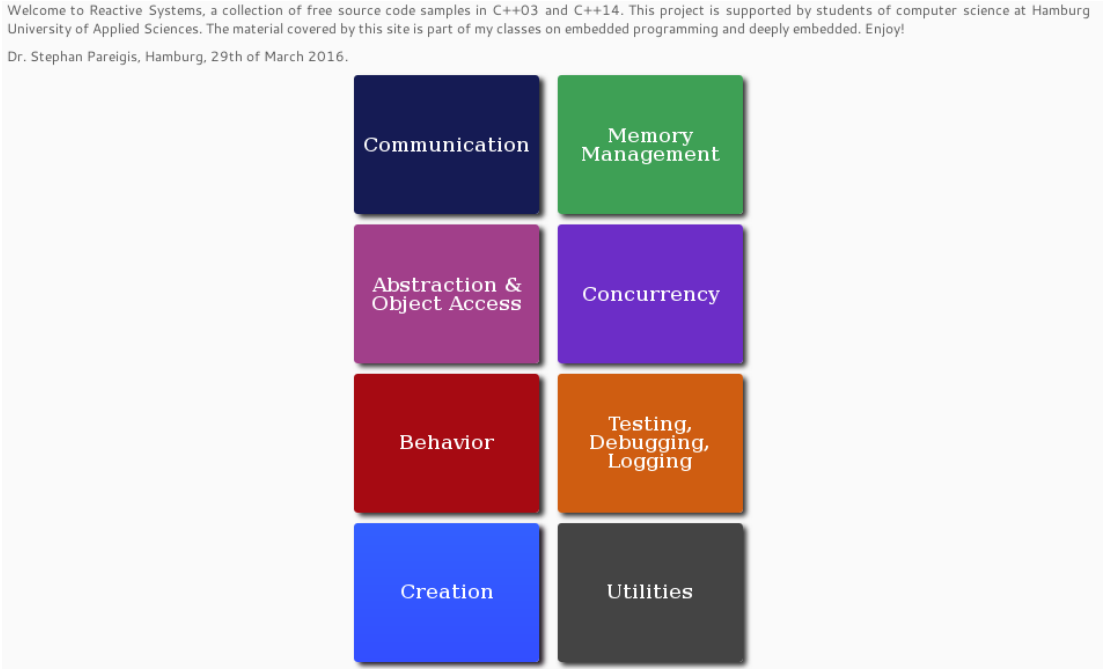

Abbildung 2.1: Startseite der Plattform

[1] Alle Bilder der Plattform sind von http://pareigis.ful.informatik.haw-hamburg.de/, Zugriffsdatum 03.04.2016

Unter diesen Kategorien gibt es wiederum Unterkategorien mit konkreten Ansätzen zur Lösung eines Problems. Hier findet man z.B. Lösungen die sich mit der Objekterzeugung auseinandersetzen. So zum Beispiel *Singleton*, *Factory*, *Abstract Factory* und *Utilities*.

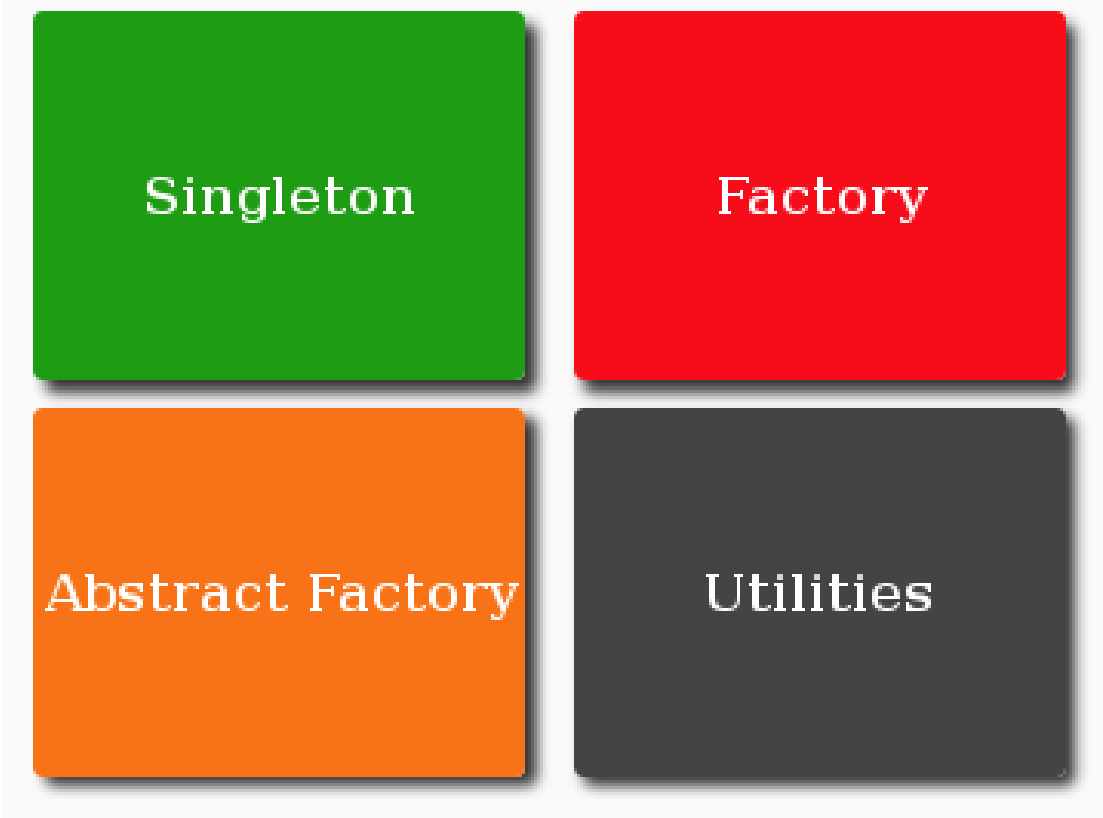

Abbildung 2.2: Auswählbare Elemente für die Objekterzeugung

Wählt man ein Element aus, so kommt man zu einer Beschreibung des Elements. Man findet z.B. eine Erklärung, Motivation und verschiedene Implementationen in verschiedenen Programmiersprachen zu den jeweiligen Elementen.

Im Folgenden ist zum Beispiel das *Factory Pattern* zu sehen.

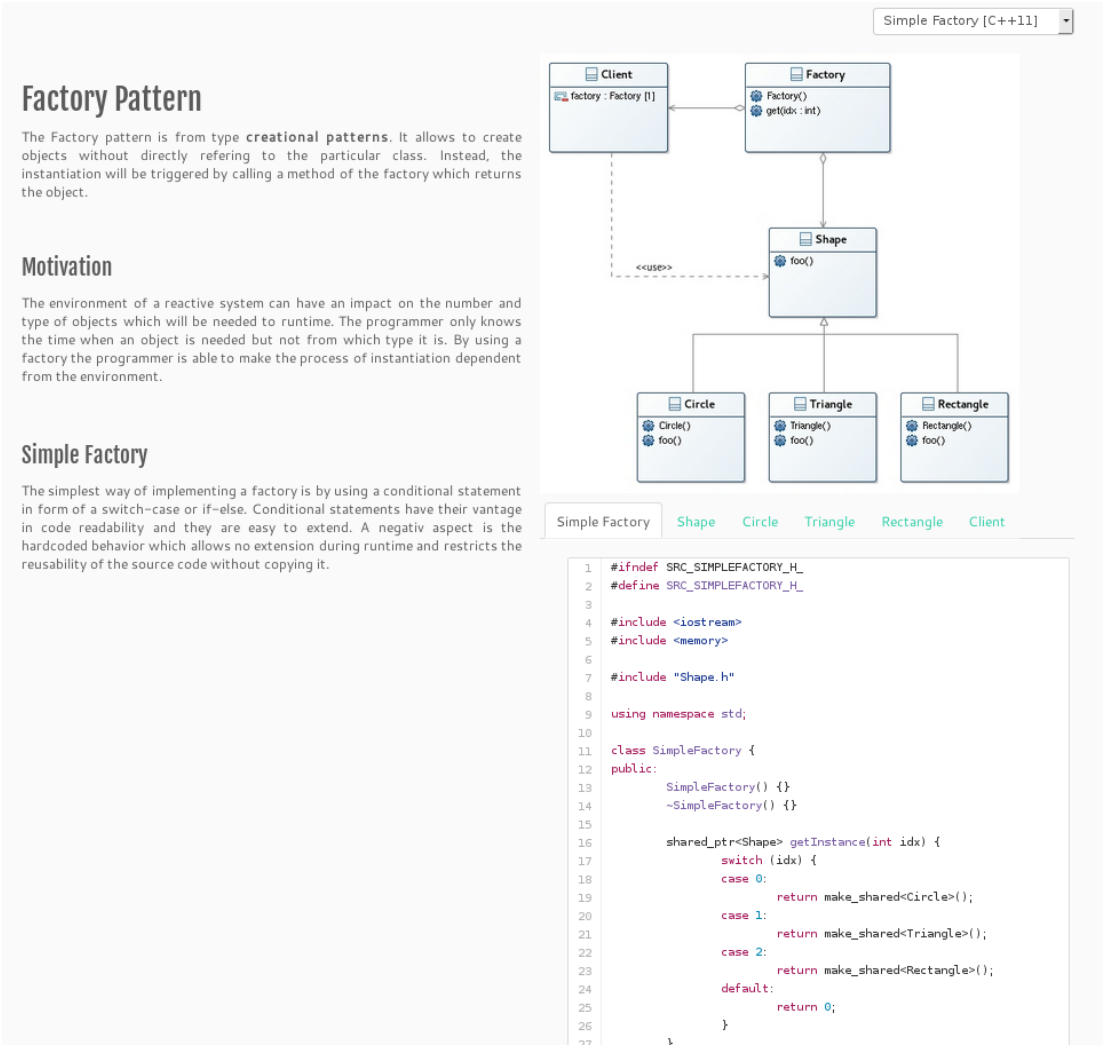

Abbildung 2.3: Die Beschreibung eines Pattern

So sieht man in Abbildung 2.3 eine Erklärung und Motivation zum Factory Pattern. Darüber hinaus gibt es zur besseren Übersicht ein Klassendiagramm zur beispielhaften Implementation. Die Implementation kann über mehrere Elemente aufgeteilt werden und dazu können beispielhafte Verwendungen der Implementation eingebunden werden. Über eine Selection Box lassen sich verschiedene Implementationen in verschiedenen Sprachen auswählen.

3 Einführung in Rust

Das folgende Kapitel beschäftigt sich mit der Einführung in die Programmiersprache Rust. Zunächst wird die Motivation der Sprache und ihre Entstehung beschrieben. Des Weiteren werden die wichtigsten Elemente der Sprache erläutert, die in den Kapiteln zur Problemlösung genutzt werden.

3.1 Was ist Rust

Rust ist eine systemnahe Programmiersprache, die den Anspruch hat so schnell wie C/C++ zu sein und darüber hinaus in vielen Bereichen Sicherheit zu schaffen. Einer dieser Bereiche ist unter anderem die parallele Programmierung mit Threads, in der der gewissenhafte Umgang mit Ressourcen wichtig ist. Sicherheit bei Speicherzugriffen will die moderne Sprache durch zentrale Konzepte wie dem Ownership Modell erreichen. Das besondere an Rust ist, dass die Sicherheit statisch zur Kompilezeit geprüft wird. Somit gibt es unter Rust keine *Segmentation Faults*, die unter C/C++ zu komplizierten und schwer diagnostizierbaren Fehlern führen können.

Rust besitzt keine Garbage Collection, was darin resultiert, dass zum einen weniger Speicher verbraucht wird, als auch eine höhere Geschwindigkeit erreichbar ist. Somit ist Rust auch für die eingebettete Programmierung prädestiniert. Programme werden mit dem in Rust geschriebenen Compiler compiliert. Der Compiler setzt auf der modularen Compiler-Infrastruktur LLVM auf, die schon andere nennenswerte Compiler wie zum Beispiel Clang (Compiler für C++) als Backend benutzen. Daher unterstützt Rust bereits Plattformen wie Windows, OS X, Linux, Android und viele mehr. Rust kategorisiert dabei den Plattformsupport in drei Verschiedene Stufen (Tier 1 bis 3), die jeweils über verschiedene Zusicherungen (z.B. Garantie des Bauens und der Lauffähigkeit) verfügen[23, Kapitel Getting Started].

Eines der Prinzipien von Rust ist die *zero cost abstraction*. Dieses sagt aus, dass die Abstraktion wie z.B. Interfaces, Closures et cetera, die die Sprache bietet, keinen Overhead in der Laufzeit und der Programmgröße bedeutet. Darüber hinaus wird alles, was zur Laufzeit nicht gebraucht wird, auch nicht in den Programmcode ausgelagert. Auch C++ arbeitet nach diesem Prinzip.[18][30] Als Beispiel seien Templates genannt, welche zur Kompilezeit für die genutzten Datentypen aufgelöst werden.

Aufgrund der aufgeführten Aspekte ist Rust auch für eingebettete Systeme sehr interessant, die gerade im Zusammenhang mit dem Thema Internet of Things immer mehr an Bedeutung gewinnen.

Die Programmiersprache Rust und ihre Konzepte entstanden 2006 als Nebenprojekt von Graydon Hoare und seit 2009 wurde die Sprache durch das Mozilla Research Team weiterentwickelt. Mittlerweile wird die Sprache durch das Rust Team, bestehend aus Mitarbeitern von Mozilla und der Rust Community, verbessert. ([22]) Am 15. Mai 2015 wurde offiziell die erste stabile Version der Sprache freigegeben. Das bekannteste in Rust geschriebene Projekt ist die Browserengine Servo von Mozilla, die auch den Ursprung der Programmiersprache darstellt und für die Weiterentwicklung von Rust genutzt wird. Rust ist im Vergleich mit C/C++ noch sehr jung und es ist abzuwarten, ob die Industrie die Sprache akzeptiert.

Für die Kompilierung und das Testen der Implementationen wurde der Compiler *rustc* in der Version 1.5 und *cargo* in der Version 0.6.0-nightly verwendet.

3.2 Hello, World!

Da es bei der Einführung in eine Programmiersprache gebräuchlich ist, hier ein *Hello, World!* Beispiel in Rust. (Siehe [23], Kapitel *Getting Started*)

Listing 3.1: *Hello, World!* in Rust

```
1 fn main() {
2     println!("Hello, {}!", "World");
3 }
```

Das Beispiel besteht aus der Definition einer *main*-Funktion, die der Startpunkt bei der Ausführung eines Rust-Programms ist. Die Funktion besteht nur aus dem Aufruf des *println*[1] Makros. Der Aufruf von Makros ist ähnlich zu dem einer Funktion. Makros werden aber mit nachfolgendem Ausrufezeichen (!) nach dem Namen des Makros aufgerufen. Das Makro *println* gibt den übergebenen *Format String* auf der Standardausgabe aus. Dabei können ähnlich zu der Funktion *printf*[2] aus C oder *String.format*[3] aus Java weitere Elemente angegeben werden, die in die Platzhalter (gekennzeichnet durch {}) des *Format Strings* eingesetzt werden. Jede Anweisung wird durch ein Semikolon abgeschlossen. Somit wird bei Ausführung des Programms der String 'Hello, World!' auf der Standardausgabe ausgegeben.

[1] Siehe https://doc.rust-lang.org/std/macro.println!.html
[2] Siehe http://www.cplusplus.com/reference/cstdio/printf/
[3] Siehe https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

3.3 Datentypen und Variablen

Rust besitzt primitive Datentypen, mit deren Hilfe komplexe Datentypen konstruiert werden können. Eine genaue Aufzählung dazu findet sich im Buch[23, Kapitel Primitive Types]. Wie die meisten Sprachen, so besitzt auch Rust Datentypen wie numerische Typen (z.B. *u32*, welches für einen unsigned, 32-Bit Integer steht), Boolean, Strings etc.

String ist ein Buffer von UTF-8 Bytes, die auf dem Heap gelagert werden. Darüber hinaus gibt es den Datentyp *&str*, der eine Referenz auf bzw. einen Ausschnitt auf einen String darstellt. Durch den Aufruf der Methode *to_owned* des Datentyps &str, wird Speicherplatz auf dem Heap reserviert und der Ausschnitt des Strings, auf den &str zeigt, auf den Heap kopiert. Somit wird der Ausschnitt des Strings geklont. Eine über Anführungszeichen erstellte Zeichenfolge (z.B. "Test") ist vom Typ &str.[22]

Variablendefinitionen werden durch das let-Schlüsselwort gekennzeichnet. Dies sagt aus, dass ein Wert an eine Variable gebunden wird. Das Format zur Definition wird durch *let mut Name: Datentyp = Wert*[4] beschrieben. Dabei beschreibt *Name* den Namen der Variable, *Datentyp* den Datentyp der Variable und *Wert* den Wert, der der Variable zugewiesen werden soll. Der Datentyp kann dabei weggelassen werden, wenn er durch den Compiler aus dem Kontext erschlossen werden kann. Wird das mut-Schlüsselwort weggelassen, so ist die Bindung immutable. Bindungen sind standardmäßig immutable, weil dies den Vorteil hat, dass Variablen nicht versehentlich verändert werden können. Daher muss man das mut-Schlüsselwort angeben, wenn man explizit ausdrücken möchte, dass eine Bindung veränderbar sein soll.[23, Kapitel Variable Bindings]

Die folgenden Beispiele verdeutlichen die Verwendung von let.

Listing 3.2: Variablen in Rust

```
1 let number: u32 = 1;
2 let vec = vec![1,2,3];
```

In Zeile 1 wird eine Variable mit dem Namen *number* definiert. Die Variable ist ein unsigned, 32-Bit Integer und besitzt den Wert 1. In Zeile 2 wird eine weitere Variable erstellt. Diese besitzt den Namen *vec*. Der Datentyp der Variable ergibt sich aus dem Rückgabewert des Aufrufs des *vec!*-Makros[5]. Der Aufruf liefert einen Vektor von signed, 32-Bit Integern zurück (Vec<i32>). Sofern der numerische Datentyp nicht aus dem Kontext erschlossen werden kann oder kein

[4] Vereinfachte Form des Formats
[5] https://doc.rust-lang.org/std/macro.vec!.html

Datentyp in der Deklaration angegeben ist, so wird dieser standardmäßig zu i32, also einem signed, 32-Bit Integer.[23, Kapitel Primitive Types] Das ist auch in diesem Beispiel der Fall.

3.4 Funktionen

Im Folgenden werden Funktionen in Rust besprochen.

Listing 3.3: Funktionen in Rust

```
1 fn square(n: i32) -> i32 {
2     let result: i32 = n * n;
3     result
4 }
```

Eine Funktion muss immer mit dem Schlüsselwort *fn* beginnen und darauf folgt der Name der Funktion. Danach kommt die Parameterliste, die durch eine geöffnete und geschlossene, runde Klammer begrenzt wird. Parameter werden im Format *Name des Parameters: Typ* angegeben. Nach der Parameterliste folgt abgetrennt durch -> der Rückgabewert der Funktion. Besitzt die Funktion keinen Rückgabewert, so muss beides weggelassen werden. Den letzten Teil bildet der Funktionskörper, der in geschweiften Klammern angegeben wird. Im Funktionskörper werden Anweisungen durch Semikolon getrennt. Der Rückgabewert ist jeweils der letzte zu erreichende Ausdruck bzw. bei Verzweigungen die letzten zu erreichenden Ausdrücke einer Funktion. In Rust ist es Konvention, das Schlüsselwort *return*, auch wenn es im Sprachumfang vorhanden ist, wegzulassen. Dieses wird nur benutzt, wenn es wirklich benötigt wird, weil zum Beispiel vorzeitig aus einer Funktion gesprungen werden muss.[23, Kapitel Functions]

3.5 Structs

Ein wichtiges Konzept, das Rust umsetzt, ist die Trennung von Daten, Implementation und Schnittstelle. Dadurch soll eine Kapselung der Implementation und somit eine Modularisierung und eine Austauschbarkeit der Implementierung gewährleistet sein. Dazu gibt es in Rust drei Schlüsselwörter. Zum einen gibt es das Schlüsselwort *struct* [23, Kapitel Structs]. Die *Struct* (nachfolgend auch Struktur) ist der *Struct* aus der Programmiersprache C/C++ sehr ähnlich. Mit Hilfe des Schlüsselworts, können komplexe Datentypen (Strukturen) erzeugt, beziehungsweise repräsentiert werden. In einer Struct wird die Mutabilität nicht angegeben, denn in Rust ist diese von der Art der Variablenbindung (gekennzeichnet durch *mut*-Schlüsselwort) abhängig. In Rust gibt es keine Klassen- bzw. Strukturvererbung.

Listing 3.4: Komplexe Datentypen in Rust

```
struct Product  {
    name: String,
    id: u32,
    cost: f32
}
```

Eine Struktur beginnt mit dem Schlüsselwort *Struct* und dem Namen der Struct. Danach folgen in geschweiften Klammern die Attribute der Struct. Diese werden im Format *Name des Attributs: Typ des Attributs* angegeben und mit Komma getrennt.[23, Kapitel Structs]

3.6 Methoden in Rust

Die Implementationen ([23], Kapitel Method Syntax) von Methoden werden in einem Block vereint, der durch das Schlüsselwort *impl* gekennzeichnet ist. Dies ist die zweite wichtige Komponente und stellt die Trennung von Implementation und Daten dar. Der Aufbau von Methoden ist identisch zu dem von Funktionen und wird daher hier nicht weiter erläutert. Es gibt kein Method-Overloading in Rust.

Listing 3.5: Implementation von Methoden in Rust

```
impl Product {
    fn new(name: &str, id: u32, cost: f32) -> Self {
        Product {name: name.to_string(), id:id, cost:cost}
    }

    fn get_id(&self) -> u32 {
        self.id
    }

    fn set_id(&mut self, id: u32) {
        self.id = id;
    }
}
```

In Rust ist es gebräuchlich einen Hilfsmethode (*new*) zur Konstruktion der Instanzen anzubieten, um die Erstellung des Objekts an einer Stelle zu bündeln und somit über einen Punkt zur Verwaltung der Erzeugung zu verfügen.

Zwei wichtige Schlüsselwörter, die im Zusammenhang der Methodenimplementation benutzt werden, sind *Self* und *self*. *Self* repräsentiert den Typ des komplexen Datentyps, für den

die Methoden implementiert werden. Das Schlüsselwort *self* steht für das Objekt, auf das diese Methode aufgerufen wird. Daher sind alle Methoden, die nicht *self* als Argument erwarten, statische Methoden, und alle, die *self* erwarten, nicht statisch. Somit ist die Methode *new* statisch, wohingegen die Methoden *get_id* und *set_id* nicht statisch sind. Statische Methoden werden mittels des ::-Operators, in dem Format *StructName::method_name(...)*, aufgerufen. Nicht statische Methoden werden über den .-Operator aufgerufen (instanz_name.methode_name(...)).[23, Kapitel Method Syntax]

Das folgende Beispiel veranschaulicht die verschiedenen Arten der Aufrufe.

Listing 3.6: Verwendung der Methoden in Rust

```
1 fn main() {
2     let mut product_a = Product::new("Produkt_A", 1, 1.50);
3
4     product_a.set_id(2);
5     println!("ID_von_Produkt_A:_{}", product_a.get_id());
6 }
```

In dem Beispiel wird ein veränderbares (*mut*) Produkt erzeugt und an die Variable *product_a* gebunden. Danach wird durch Aufruf der *set_id* Methode die ID des Produkts geändert und in der darauffolgenden Zeile wird die ID mit Hilfe der *get_id* auf der Standardausgabe ausgegeben.

3.7 Generics

Rust verfügt über Generics ([23], Kapitel Generics), die ähnlich zum Template-Konzept aus C++ oder auch dem Generic-Konzept von Java sind. Durch diese können Redundanzen im Quellcode vermieden werden, indem z.B. eine Funktion generisch geschrieben wird. Generics können in Funktionen/Methoden, Structs, Enums, Traits und den Implementationen benutzt werden.

Generics werden, wie in vielen anderen Programmiersprachen, in spitze Klammern (<...>) gechlossen. Für die Variable sollte ein, in dem Kontext passender, Name gewählt werden. Ansonsten ist die Variable *T* gebräuchlich, die auch in dem folgenden Beispiel genutzt wird. Zusätzlich werden nach einem Doppelpunkt die Typen, die von dem generischen Typ implementiert werden müssen, mit + getrennt angegeben (*T: Typ1 + Typ2 + ... + TypN*). Dadurch wird eine Typkomposition ermöglicht.[23, Kapitel Generics]

Listing 3.7: Generics in Rust

```
1 struct Message<T: Clone> {
```

```
2     obj: T,
3 }
4
5 impl<T:Clone> Message<T> {
6     fn new(obj: T) -> Self {
7         Message {obj: obj}
8     }
9
10    fn get_object(&self) -> T {
11        self.obj.clone()
12    }
13 }
14
15 fn main() {
16    let msg = Message::new("Test");
17    let obj = msg.get_object();
18    println!("Object of Message: {}", obj);
19 }
```

Die beispielhafte Struct *Message* besitzt einen generischen Typ *T*. Der Typ *T* muss zusätzlich noch die Eigenschaft *Clone* implementieren, welche aussagt, dass T geklont werden kann. Message besitzt ein Attribut *obj* vom Typ *T*. In den Zeilen 5 bis 13 wird die Struct implementiert. Es wird eine Konstruktormethode und eine Methode *get_object* implementiert, die einen Klon des in der Message gekapselten Objekts zurückgibt. In der *main* Methode wird eine beispielhafte Verwendung skizziert. Zunächst wird eine Message-Instanz erzeugt, die einen String 'Test' beinhaltet. Danach wird die *get_object* Methode aufgerufen und der zurückgegebene Klon an eine Variable gebunden. Das ganze wird danach auf der Standardausgabe ausgegeben.

Die *Generics* im Zusammenspiel mit den nachfolgend dargestellten *Traits* bilden den Kern der Polymorphie in Rust.

3.8 Traits

Traits (dt. Eigenschaft, Charakterzug oder Merkmal) in Rust sind die Analogie zu Schnittstellen aus Java oder C++ und stellen die dritte wichtige Komponente zur Programmierung in Rust dar. Implementiert eine Struct einen Trait, so sagt dies aus, dass die Struktur eine bestimmte Eigenschaft besitzt. [23, Kapitel Traits]

Rust richtet sich nach dem *Composition over Inheritance*-Prinzip (Siehe [7], S.27), welches aussagt, das Objektkomposition der Klassenvererbung vorgezogen werden soll. Ein Nachteil

dieses Prinzips ist, dass alle Methoden, die eine Schnittstelle anbietet auch von der implementierenden Struktur implementiert werden müssen. Rust löst dieses Problem, indem die Schnittstellen (Traits) selber Implementationen anbieten können, auch *Default Implementationen* genannt. Darüber hinaus bietet Rust Vererbung in Schnittstellen an, auf die hier aber nicht weiter eingegangen wird.[23, Kapitel Traits]

3.8.1 Aufbau eines Traits

Der Aufbau eines Traits hat Ähnlichkeit zu der Implementationen von Methoden durch *impl*. Der einzige Unterschied ist, dass die Implementation einer Methode optional ist und somit auch nur eine Methodensignatur angeben werden kann. Die eigentliche Implementation muss dann von den jeweiligen Strukturen, die einen Trait umsetzen, implementiert werden.

Das folgende Beispiel zu Traits bezieht sich auf das Produkt aus den vorherigen Kapiteln.

Listing 3.8: Einführung eines Traits

```
impl Product {
    fn new(name: &str, id: u32, cost: f32) -> Self {
        Product {name: name.to_string(), id:id, cost:cost}
    }
}

trait HasId {
    fn get_id(&self) -> u32;
    fn set_id(&mut self, u32);

    fn print_id(&self) {
        println!("ID:_{}", self.get_id());
    }
}

impl HasId for Product {
    fn get_id(&self) -> u32 {
        self.id
    }

    fn set_id(&mut self, id: u32) {
        self.id = id;
    }
}
```

```
25
26 fn main() {
27     let mut product_a = Product::new("Produkt_A", 1, 1.50);
28
29     product_a.set_id(2);
30     product_a.print_id();
31 }
```

Durch die Einführung des *HasId*-Traits werden die Methoden *get_id* und *set_id* aus der Methodenimplementation entfernt. Stattdessen werden in Zeile 8 bis 9 die Methodensignaturen der beiden Methoden in dem neu erstellten Trait angegeben. Darüber hinaus wird eine Methode *print_id* implementiert. Diese gibt die ID der Instanz zurück. Dabei greift sie auf die Methode *get_id* zu. Danach wird der *HasId*-Trait in den Zeilen 16 bis 24 durch die *Product*-Struktur implementiert. Die Implementation ist äquivalent zu der Implementation aus dem *Methoden in Rust*. Auch der beispielhafte Aufruf in der Funktion *main* hat sich nicht geändert.

3.8.2 Statischer und dynamischer Dispatch

Rust unterscheidet zwischen statischem und dynamischen Dispatch ([23], Kapitel *Trait Objects*). Bei statischem Dispatch führt Rust eine sogenannte *Monomorphization* durch. Dies bedeutet, dass Rust bei der Kompilierung die generischen Implementationen von Funktionen und Methoden durch die echten Typen ersetzt, indem es für jeden Typ eine extra Implementation generiert. Dadurch ist Inlining möglich, aber gleichzeitig wird auch die Größe des Quellcodes erhöht. Auch C++ führt dies aufgrund des *zero-overhead*-Prinzips durch[18]. Darüber hinaus verfügt Rust auch über dynamischen Dispatch. Dabei benutzt Rust sogenannte *Trait Objects*, die über eine *Virtual Table* (kurz *vtable*) arbeiten. Daher besitzen diese einen Zeiger auf eine eine Tabelle mit den jeweiligen Methoden eines Traits. Dabei müssen Zeiger für die Repräsentation der Trait Objects benutzt werden, weil die Größe des Objekts, das einen Trait implementiert, variieren kann. Darüber hinaus wird eine sogenannte *type erasure* durchgeführt. Dadurch wird das Wissen des Compilers über den eigentlichen Typ des Objekts gelöscht.[23, Kapitel Trait Objects]

Die Verwendung von Trait Objects wird unter anderem in dem Kapitel über das Builder Pattern aufgezeigt.

3.9 Ownership

Eines der größten Probleme, im Kontext der Ausfallsicherheit in Programmen, stellt bei den marktführenden Programmiersprachen im Bereich der hardwarenahen Programmierung immer noch direkter Speicherzugriff dar. Durch neue Standards wie C++11, in dem die schon zuvor lange in Boost enthaltenen Smart Pointer in C++ eingeführt wurden, sollen die Programme einfacher und sicherer gestaltet werden können. Allerdings gibt es weiterhin Anwendungsfälle, in denen Probleme auftreten, die nur mit sauberer Programmierung oder Analysetools (siehe zum Beispiel Valgrind[4]) verhindert werden können.

3.9.1 Smart Pointer in C++

In C++11 wurden Smart Pointer eingeführt, die das Programmieren mit dynamischem Speicher erleichtern und sicherer machen sollen. Allerdings muss man weiterhin bei der Programmierung mit Smart Pointern aufpassen, wie das nachfolgende Beispiel aufzeigt.

Listing 3.9: C++ unique_pointer [4]

```
#include <iostream>
#include <memory>

using namespace std;

int main ()
{
        unique_ptr<int> orig(new int(5));

        cout << *orig << endl;

        auto stolen = move(orig);

        cout << *orig << endl;
}
```

In dem Beispiel wird zunächst ein unique_pointer, der auf einen Integer zeigt, welcher wiederum auf dem Heap liegt, erzeugt und danach über diesen unique_pointer auf den Integer zugegriffen beziehungsweise der Wert des Integers auf der Standardausgabe ausgegeben.

[4] *Siehe http://valgrind.org/*

Danach wird ein neuer unique_pointer erzeugt, der über das *move*-Keyword alleinigen Zugriff auf den Integer erhält. Jetzt wird wieder auf den ersten, erzeugten Pointer zugegriffen, was in einem Segmentation Fault zur *Laufzeit* resultiert, da dieser nach dem *move* keinen Zugriff mehr auf den Integer hat. Durch *nullptr*-Checks ist ein abfangen dieses Fehlers zur Laufzeit möglich.

Rust versucht mit dem Ownership Modell solche Fehler zu verhindern.

3.9.2 Das Ownership Modell in Rust

Das Ownership Modell von Rust stellt eines der zentralen Konzepte der Programmiersprache dar. Es sagt aus, dass eine Ressource immer nur genau einen Besitzer hat. Endet die Lebenszeit des Besitzers, so endet auch die Lebenszeit der Ressource.[23, Kapitel Ownership] Die Ressource wird automatisch freigegeben, wenn der Scope, in dem der Besitzer liegt, freigegeben wird. Ähnlich funktioniert dies in C++, wo beim Abbauen des Stacks der Destruktor der lokalen Variablen aufgerufen wird. Bei Ressourcen auf dem Heap muss der Entwickler allerdings selber die Speicherreservierung und -freigabe verwalten. Eine Abhilfe bieten dabei die eben gezeigten Smart Pointer.

Im Folgenden wird das eben in C++ skizzierte Beispiel in Rust übersetzt.

Listing 3.10: Das Ownership Modell von Rust [4]

```
fn main() {
        let orig = Box::new(5);

        println!("{}", *orig);

        let stolen = orig;

        println!("{}", *orig);
}
```

Auch hier wird zunächst ein Zeiger auf einen Integer, der wiederum auf dem Heap liegt, erzeugt. Dies geschieht über den Aufruf von *Box::new*[6], welches Speicher auf dem Heap für ein übergebenes Objekt reserviert und dieses dann dort platziert. Standardmäßig, also ohne

[6] https://doc.rust-lang.org/std/boxed/struct.Box.html

Benutzung von Box, werden Objekte auf dem Stack erzeugt. Danach wird der Wert des Integers auf der Standardausgabe ausgegeben. Jetzt wird eine weitere Variable erzeugt, die durch die Zuweisung das Ownership, also den alleinigen Besitz über den Integer erhält. Somit führt der erneute Zugriff auf den zuerst erzeugten Pointer zu folgendem Fehler *error: use of moved value: ‘*orig‘*, der aber zur *Kompilezeit* ausgegeben wird!

3.9.3 Ein komplexeres Beispiel

Je komplexer und abstrakter die Projekte sind, desto höher ist die Wahrscheinlichkeit, Fehler zu machen. Das nachfolgende Beispiel zeigt die Gefahr der Verwendung von Smart Pointern im Zusammenhang mit Containern.

Listing 3.11: C++ Smart Pointer in Containern [4]

```
#include <vector>
#include <memory>
#include <iostream>

using namespace std;

int main() {
        vector<unique_ptr<int>> v;
        v.push_back(make_unique<int>(5));

        cout << *v[0] << endl;

        auto pointer_to_5 = move(v[0]);
        cout << *pointer_to_5 << endl;

        cout << *v[0] << endl;
}
```

Zunächst wird ein Vektor von unique_pointern erstellt. Danach wird ein unique_pointer, der auf einen Integer zeigt, in den Vektor gepusht. Jetzt wird auf den Wert des ersten Elements, welcher dem Integer entspricht, zugegriffen und dieser auf der Standardausgabe ausgegeben. Im Folgenden wird ein neuer unique_pointer erzeugt, der durch das *move*-Keyword alleiniger Besitzer des zuvor erzeugten und im Vektor liegenden Integers ist. Somit führt die Dereferenzierung in der darauffolgenden Zeile zu einem Segmentation Fault.

Das Beispiel in Rust übersetzt sieht folgendermaßen aus.

Listing 3.12: Rust Raw Pointer in Containern [4]

```
fn main() {
        let v = vec![Box::new(5)];

        println!("{}", *v[0]);

        let pointer_to_5 = v[0];

        println!("{}", *pointer_to_5);

        println!("{}", *v[0]);
}
```

Auch hier wird zunächst ein Vektor erstellt und bei der Erzeugung direkt ein Zeiger auf einen, auf dem Heap liegenden, Integer gepusht. Danach wird der Wert des Integers auf der Standardausgabe ausgegeben. Jetzt wird ein neuer Zeiger erstellt, der das Ownership über das erste Element des Vektors erhält. Im Folgenden wird der Wert der beiden Zeiger ausgegeben. Allerdings lässt sich dieses Programm gar nicht compilieren. Dabei wird die Fehlermeldung *error: cannot move out of indexed content* ausgegeben.

3.9.4 Ein gutes Konzept?

Rusts Ownership Modell stellt einen interessanten Ansatz dar, wie mit Ressourcen umgegangen werden kann. Fehler wie Segmentation Faults werden schon zur Kompilezeit gefunden. Gerade in Embedded Systemen, die möglichst selbstständig und unabhängig von der Außenwelt laufen sollen, bietet dieser Ansatz Sicherheit. Auf der anderen Seite muss man sich mit den verschiedenen Lebenszeiten und der Verschiebung des Zugriffs auf Ressourcen auseinandersetzen.

Neben dem alleinigen Besitz einer Ressource kommt noch dazu, dass man Ressourcen verleihen (*Borrowing*) kann. Dies geschieht mit Hilfe von Referenzen. Dazu werden Variablen mit *&* annotiert. Das korrekte Verleihen von Ressourcen prüft Rust auch, denn die Ressource kann nur so lange verliehen werden, wie man selber existiert. Gleichzeitig muss eine geliehene Ressource mindestens so lange leben, wie man selber lebt. Dazu werden Referenzen und Strukturen mit sogenannten *Lifetimes*[23, Kapitel Lifetimes] annotiert, die der Compiler nutzt,

um sicherzustellen, dass Ressourcen lange genug leben. Darüber hinaus kann eine Ressource entweder lesend von mehreren Variablen oder schreibend von nur genau einer Variable genutzt werden.[23, Kapitel References and Borrowing] Eine Verwendung findet sich im Kapitel zum Builder Pattern.

Jede neue Sprache besitzt Schwierigkeiten beim Einstieg und gerade in der hardwarenahen Programmierung ist gewissenhafte Programmierung notwendig. Rust schafft mit dem Ownership Modell eine gute Grundlage für genau diese Art der Programmierung.

3.10 Fehlerbehandlung

Rusts Fehlerhandlung ([23], Kapitel *Error Handling*) verzichtet auf Exceptions, die in Sprachen wie Java und C++ vorkommen, und führt eine Fehlerbehandlung über Typen durch ([22], *Error Handling*). Dabei bilden der Typ *Option* und der Typ *Result* den Kern dieses Konzepts. Beide Typen sind Enums. Diese sind relativ ähnlich zu den Enums aus C++ und werden hier nicht weiter beschrieben. Enums in Rust sind eine Art *Tagged Union*[7]. Sie verbrauchen also immer nur so viel Speicherplatz wie die größte Variante des Enums.[14]

3.10.1 Option

Gibt eine Funktion ein Datentyp *Option* zurück, so signalisiert das, dass der Rückgabewert optional sein kann.

Listing 3.13: Deklaration von Option

```
enum Option<T> {
    Some(T),
    None,
}
```

Der Typ *Option* benutzt Generics (Generischer Typ *T*), um einen variablen Typ zu kapseln. Soll ein Objekt zurückgegeben werden, so wird dieses in *Some(T)* gekapselt. Soll signalisiert werden, dass kein Objekt zurückgegeben wird, so wird *None* zurückgegeben.[23, Kapitel Error Handling]

Eine beispielhafte Verwendung stellt die get-Methode[8] von HashMap dar. Die Signatur sieht folgendermaßen aus *fn get(&self, k: &Q) -> Option<&V>*. Die Methode erwartet eine Referenz

[7] Siehe https://en.wikipedia.org/wiki/Tagged_union
[8] Signatur vereinfacht dargestellt, https://doc.rust-lang.org/std/collections/struct.HashMap.html

auf den Key *k*, dass den generischen Typ Q hat, und liefert, in Option gekapselt, eine Referenz auf ein Objekt vom generischen Typ V. Das heißt, lässt sich ein Schlüssel k auf einen Wert mappen, so wird dieser in *Some* gekapselt zurückgegeben (erste Variante von Option). Ist kein Wert vorhanden, so wird None zurückgegeben. Also vereinfacht dargestellt, wenn es zu dem Schlüssel einen Wert gibt, wird dieser gekapselt zurückgegeben. Ansonsten wird durch das *None* signalisiert, das kein Wert vorhanden ist.

Option bietet die Methode *unwrap*[27] an. Ist die Variante des Option-Typs *Some*, so liefert sie die in Option gekapselte Instanz zurück. Ist dagegen die Variante *None*, so wird das Makro *panic!* aufgerufen, das die Ausführung des Threads, in dem das Makro aufgerufen wird, stoppt. Dabei wird eine generische Fehlermeldung ausgegeben. Dies sollte nur genutzt werden, sofern dieser Effekt gewünscht ist. Besser ist es stattdessen den Rückgabewert zu verarbeiten und anhand dieses Werts weitere Aktionen durchzuführen.

3.10.2 Result

Der Datentyp *Result* bietet darüber hinaus die Möglichkeit zu signalisieren, ob während der Ausführung ein Fehler aufgetreten ist, der dann zurückgegeben werden kann.

Listing 3.14: Deklaration von Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Result besitzt zwei generische Typen (*T* und *E*). T steht dabei wieder den Typ des gekapselten Objekts (*Ok(T)*) und E steht für den Typ eines Fehlerobjekts (*Err(E)*), das zurückgegeben wird.[23, Kapitel Error Handling]

Der Entwickler wird durch diese Art der Fehlerbehandlung gezwungen, einen Rückgabewert auf seine unterschiedlichen Varianten zu überprüfen. Dadurch ist eine Dereferenzierung eines invaliden Objekts nicht möglich. Darüber hinaus kann man anhand des Rückgabewerts ablesen, wie eine Funktion/Methode arbeitet, also z.B. ob eine Funktion zu einem Fehler/nicht vorhandenem Objekt führen kann. Dies verbessert die Lesbarkeit des Quellcodes.

3.11 Macros

Rust verfügt über ein Makrosystem ([23], Kapitel *Macros*), das aber im Gegensatz zu C/C++ auf dem *abstrakten Syntax Baum* (engl. abstract syntax tree, kurz AST) arbeitet und nicht mit Hilfe von Textersetzung (Siehe [6]). Makros erlauben die Abstraktion auf einem syntaktischen Level. Makros sind schwerer zu lesen, zu debuggen und zum Teil eng an andere Implementationen gekoppelt. Daher sollten Makros immer mit Vorsicht verwendet werden.

Makros können über mehrere Wege definiert werden, aber im Folgenden wird nur der Weg über das Schlüsselwort *macro_rules!* veranschaulicht.

Listing 3.15: Aufbau eines Makros in Rust

```
macro_rules! macro_name {
    (first_pattern) => {
        statement_one;
        ...
        statement_n;
    };

    (second_pattern) => {
        statement_one;
        ...
        statement_n;
    }
}
```

Nach dem Schlüsselwort und dem Namen des Makros folgt der Körper. Dieser besteht im Grunde genommen aus Patterns und Anweisungsblöcken. Bei der Kompilierung wird schon in einer frühen Phase das Makro *expanded*. Dies bedeutet, dass der Knoten im AST, in dem das Makro sitzt, durch die Anweisungen des Makros ersetzt wird. Erst danach wird überprüft, syntaktische und semantische Fehler in der Implementierung vorliegen. Somit ist eine sichere Implementation des Makros gewährleistet. In den Patterns kann man Variablen festlegen, an die die übergebenen Argumente gebunden werden. Darüber hinaus können Wiederholungen definiert werden, die Ähnlichkeit zur Kleenneschen Hülle aufweisen [10, S.95]. Die Implementation geht auf Kohlbecker zurück[12]. Für eine Auflistung der Möglichkeiten siehe [23, Kapitel Macros].

Die folgende Definition eines Makros zur Erzeugung von Maps soll die Implementation und Verwendung von Makros veranschaulichen. Dazu soll eine Map in der Form *create_map(key1 => value1, ..., keyN => valueN)* erzeugt werden können.

Listing 3.16: Definition des Makros

```
macro_rules! create_map {
    ( $( $key:expr => $value:expr),* ) => {
```

Das Makro verfügt über ein Pattern. In dem Pattern wird alles von der Kleenneschen Hülle umschlossen ($(...)*). In der Kleenschen Hülle wird zunächst eine Expression (*expr*) gefordert, die an die Variable *$key* gebunden wird. Danach folgt die Zeichenfolge => und eine weitere Expression, die an die Variable *$value* gebunden wird. Wenn dieses Pattern beim Aufruf des Makros gefunden wird, so wird der folgende rechte Teil in den Knoten expandiert.

Listing 3.17: Körper des Makros

```
        let mut temp_map = HashMap::new();
        $(
            temp_map.insert($key, $value);
        )*
        temp_map
    };
}
```

Zunächst wird eine leere *HashMap* erzeugt. In diese werden dann durch die Kleenesche Hülle ($(...)*) und mittels der Methode *insert*, die die Map anbietet, die Wertpaare, die an *$key* und *$value* gebunden sind, eingefügt. Am Schluss wird dann die gefüllte Map zurückgegeben. Nach der Implementation kann das Makro folgendermaßen verwendet werden.

Listing 3.18: Verwendung des Makros

```
1   #[test]
2   fn test_macro() {
3       let costs = create_map!(
4           "Product1" => 1.5,
5           "Product2" => 3.5,
6           "Product3" => 0.5
7       );
8
9       assert_eq!(Some(&1.5), costs.get("Product1"));
10      assert_eq!(Some(&3.5), costs.get("Product2"));
11      assert_eq!(Some(&0.5), costs.get("Product3"));
12  }
```

In Zeile 3 bis 7 wird die Map über das Makro mit Wertepaaren gefüllt und danach wird in den Zeilen 9 bis 11 überprüft, ob die Elemente richtig in die Map eingefügt wurden.

Makros bieten eine mächtige Form zur Abstraktion und Vermeidung von Redundanzen, die zur Präsentation in den nachfolgenden Implementationen illustriert werden soll.

3.12 Attributes

Attributes werden dafür genutzt, Deklarationen (Funktionen, Strukturen etc.) in Rust zu annotieren.[23, Kapitel Attributes] Dadurch können Aktionen, z.B. durch den Compiler, durchgeführt werden. Attribute werden in der Form *#[AttributName(Argument1,..,ArgumentN)]* angegeben, wobei die Angabe von Argumenten je nach Attribut optional sein kann. Erwartet ein Attribut keine Argumente, so werden auch die runden Klammern um die Argumente weggelassen.

Ein Beispiel ist das *derive*-Attribut. Dieses erlaubt die automatische Generierung von Implementationen von bestimmten Traits für Strukturen.[24]

Listing 3.19: Verwendung des derive-Attributes

```
#[derive(Clone)]
struct Product  {
        name: String,
        id: u32,
        cost: f32
}
```

In diesem Beispiel ist die Product-Struktur mit dem derive-Attribut annotiert und als Argument wurde *Clone* übergeben. Dadurch generiert der Compiler bei der Kompilation eine Implementation des Clone-Traits für die Product-Struktur.

Ein anderes Beispiel ist das *test*-Attribut. Mit Hilfe dieses Attributes können Funktionen annotiert werden und dadurch wird dem Compiler signalisiert, dass eine Funktion zum Testen der Implementation gedacht ist. Somit wird diese Funktion bei der Kompilation ausgelassen bzw. nur bei der Durchführung der Tests ausgeführt. [24]

Ein Beispiel zur Verwendung des test-Attributes findet sich in der Erklärung zum Testen von Implementationen in Rust im nächsten Abschnitt.

3.13 Tests

Rust verfügt über die Möglichkeit zur Implementation von Tests, welche durch das *test-driven Development* noch wichtiger geworden sind. Dies funktioniert ohne zusätzliche Bibliotheken.

Dazu wird das test-Attribut benutzt, das im vorherigen Teil erklärt wurde. Darüber hinaus wird das *cfg*-Attribut genutzt. Durch dieses Attribut ist eine bedingte Kompilierung möglich.[24]

Listing 3.20: Beispiel eines Testmoduls (von [23], Kapitel Testing)

```
1 pub fn add_two(a: i32) -> i32 {
2     a + 2
3 }
4
5 #[cfg(test)]
6 mod tests {
7     use super::add_two;
8
9     #[test]
10     fn it_works() {
11         assert_eq!(4, add_two(2));
12     }
13 }
```

In Zeile 1 bis 3 wurde zunächst eine simple Funktion *add_two* implementiert, die den Wert 2 zu der Variable a, welche vom Typ i32 ist, addiert und zurückgibt. Diese Funktion soll auf ihre Korrektheit überprüft werden.

Rust verfügt über die Möglichkeit, Projekte über das Schlüsselwort *mod* zu modularisieren. Die genau Verwendung wird nicht weiter beschrieben, aber im offiziellen Buch über Rust findet sich eine ausführliche Beschreibung dazu.[23, Kapitel Crates and Modules] In diesem Beispiel wird in Zeile 6 bis 13 ein Modul *tests* definiert, das mit dem Attribut *#[cfg(test)]* annotiert ist. Dies führt dazu, dass bei normalen Bauen des Projekts das ganze Modul vom Compiler ignoriert wird. Damit die Tests auch kompiliert werden, muss der Compiler mit einem test-Flag ausgeführt werden.

In Zeile 7 wird über die *use*-Anweisung der Pfad gekürzt, über den die add_two-Funktion aufgerufen werden muss. Da sie im übergeordneten Bereich liegt, muss sie über super::add_two genutzt werden. Durch Verwendung von use kann die Funktion nun direkt über add_two aufgerufen werden. In Zeile 10 bis 12 wird eine Funktion *it_works* definiert. Über das *assert_eq!*-Makro[9] wird geprüft, ob bei Aufruf der Funktion add_two mit dem Wert 2 der zu erwartende Wert 4 zurückgegeben wird. Die Funktion ist mit dem Attribut test annotiert, um dem Compiler zu signalisieren, dass es sich um eine Test-Funktion handelt.

Tests können unter anderem über den Package Manager *Cargo* ausgeführt werden. Dies wird im nächsten Abschnitt beschrieben.

[9] https://doc.rust-lang.org/std/macro.assert_eq!.html

3.14 Cargo

Cargo (siehe für genau Installation und Verwendung [21]) ist ein Package Manager, welcher die Verwaltung von Abhängigkeiten vereinfachen soll. Dazu baut es auf dem Rust Compiler auf. Er verfügt über viele Kommandos, die das Bauen und Testen von Rust Projekten vereinfachen sollen. Die in dieser Arbeit implementierten Projekte wurden mit Cargo gebaut und getestet. Cargo wird, genau so wie Rust selber, auch durch das Rust Team gepflegt.[21]

Zum Testen der Projekte muss im Pfad des Projekts *cargo test* auf der Kommandozeile aufgerufen werden. Danach sollten die mit *#[test]*) annotierten Funktionen ausgeführt und das Ergebnis auf der Kommandozeile ausgegeben werden.

4 Erzeugungsmuster

Erzeugungsmuster sollen die Erzeugung von Objekten, sowohl zur Kompile- als auch zur Laufzeit, kapseln und die Schritte zur Erzeugung eines Objekts in die zuständigen Komponenten verschieben, um eine höchstmögliche Abstraktion und Modularisierung zu erzeugen. [7, S.101]

4.1 Builder Pattern

Das Builder Pattern wird durch die Gang of Four folgendermaßen beschrieben[7, S.119]:

> Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.

Die folgende Abbildung stellt den Aufbau der Produkterzeugung durch das Builder Pattern dar.

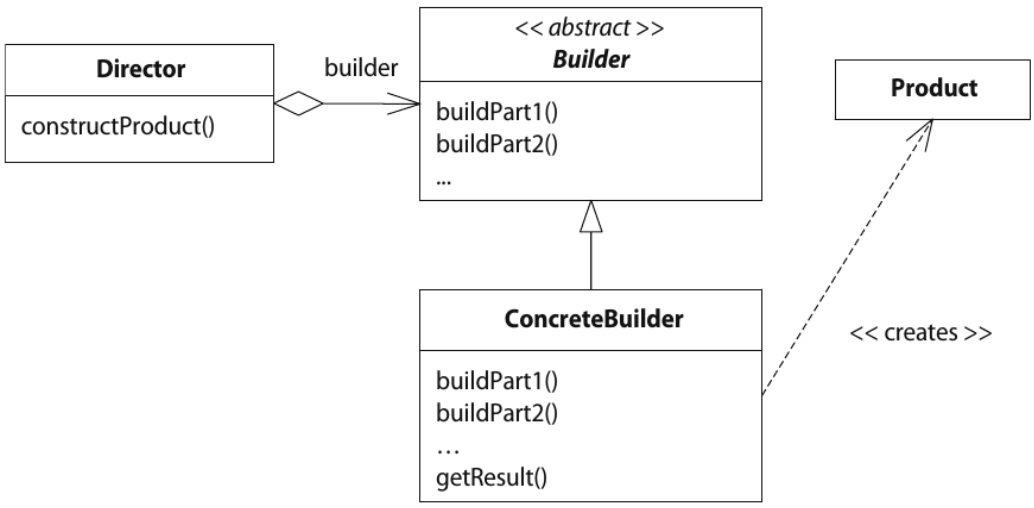

Abbildung 4.1: Das Builder Pattern nach [5], S.31

Das Builder Pattern enthält einen Director. Diesem wird der gewünschte Builder übergeben Der Director stößt die Produktion und einzelnen Schritte zur Fertigstellung des Produkts an. Der Builder ist die Schnittstelle, die vom Director angesprochen wird. Diese umfasst die eigentlichen Produktionsschritte, die zur Realisierung des Produkts nötig sind. Der ConcreteBuilder stellt eine konkrete Umsetzung der Schnittstelle des Builders dar und ist für die eigentliche Produktion des Produkts zuständig.[7, S.119ff]

4.1.1 Umsetzung in Rust

Im Folgenden wird das Builder Pattern anhand der Produkterzeugung veranschaulicht. Die Implementation wird der Übersicht wegen aufgeteilt.

Zunächst definieren wir die Schnittstelle des Builder über einen Trait.

Listing 4.1: Der Builder

```
trait  Builder {
    fn init_product(&mut self);
    fn calculate_costs(&mut self);
    fn get_product(&self) -> Box<Product>;
}
```

Der Builder bietet eine Operation zur Initialisierung des Products (*init_product*) und zur Berechnung der Kosten des Produkts (*calculate_costs*) an. Am Ende kann man das Produkt mittels einer weiteren Operation (*get_product*) empfangen.

Der Director nutzt diese Schnittstelle zur Kommunikation mit dem konkreten Builder.

Listing 4.2: Der Director

```
struct Director<'a>{
    builder: &'a Builder,
}

impl<'a> Director<'a> {
    fn new(product_builder: &mut Builder) -> Director
     {
        product_builder.init_product();
        product_builder.calculate_costs();
        Director{builder: product_builder}
    }

```

```
13      fn get_product(&self) -> Box<Product> {
14          self.builder.get_product()
15      }
16  }
```

Der Director besitzt eine Referenz auf einen Builder. Dieses Trait Object (*builder: &'a Builder*) wird ihm bei der Konstruktion des Objekts übergeben (Methode *new*, Definition Zeile 6 bis 11). Ein Trait Object erkennt man daran, dass es eine Referenz (&) auf einen Trait ist (hier *Builder*). Die Größe und der eigentliche Typ des Objekts, dass sich hinter diesem Trait befindet, sind unbekannt und daher wird ein Zeiger benötigt. Darüber hinaus ist die Struktur mit sogenannten Lifetimes annotiert. Diese befinden sich in spitzen Klammern (in diesem Fall *<'a>*) bei der Definition der Struktur (Zeile 1) und in der Definition des impl-Blocks (Zeile 5). Des Weiteren wird die Referenz auf den Builder annotiert (*builder: &'a Builder*). Das heißt, es gibt eine Lifetime *'a* und diese wird vom Compiler benötigt, um sicherzustellen, dass ein referenziertes Objekt (hier der Builder) mindestens so lange lebt, wie das Objekt, dass die Referenz besitzt (hier der Director). Daher kann es nicht dazu kommen, dass man auf einen invaliden Speicherbereich zugreift, weil ein Objekt bereits wieder freigegeben wurde. Ein use-after-free Fehler[3] kann somit nicht auftreten.

Bei der Erstellung des Builders wird das Produkt initialisiert (*init_product*) und weitere Produktionsschritte können angestoßen werden (hier z.B. *calculate_costs*). Darüber hinaus besitzt der Director eine Methode zur Herausgabe des Produkts (*get_product*).

Die konkrete Implementierung des Builders, auf die der Director eine Referenz hat, sieht wie folgt aus.

Listing 4.3: Konkrete Implementierung des Builders

```
1  struct ConcreteBuilder {
2      product: Box<Product>
3  }
4
5  impl ConcreteBuilder {
6      fn new() -> ConcreteBuilder {
7          ConcreteBuilder{product: Box::new(Product::new())}
8      }
9  }
10
11 impl Builder for ConcreteBuilder {
12     fn init_product(&mut self) {
13         self.product = Box::new(Product::new());
```

```
14      }
15
16      fn calculate_costs(&mut self) {
17          self.product.cost = 2.0;
18      }
19
20      fn get_product(&self) -> Box<Product> {
21          self.product.clone()
22      }
23  }
```

Der konkrete Builder enthält ein Heap liegendes Produkt (*Box<Produkt>*). Das Produkt wird bei der Erzeugung des konkreten Builder auf dem Heap erzeugt (*new*). Der konkrete Builder implementiert die Methoden des Builder Traits (*init_product, calculate_costs und get_product*). Die beispielhaften Implementationen für init_product erzeugt ein neues Produkt und calculate_costs berechnet und setzt die Kosten des Produkts.

Das auf dem Heap liegende Produkt wird mit Hilfe der Methode *get_product* zurückgegeben. Dabei wird das Produkt über Aufruf der *clone*-Methode geklont und der Klon zurückgegeben.

Listing 4.4: Das Produkt

```
1  #[derive(Clone)]
2  struct Product {
3      name: String,
4      id: u32,
5      cost: f32,
6  }
7
8  impl Product {
9      fn new() -> Product {
10         Product{name: "".to_owned(), id: 0, cost: 0.0}
11     }
12 }
```

Das Produkt besitzt seine spezifischen Attribute (hier z.B. Name, ID und Kosten). Darüber hinaus wird über das derive-Attribut in Zeile 1 der Clone-Trait für die Product-Struktur implementiert.

4.1.2 Test der Implementation

Im folgenden eine Implementierung eines einfachen Testcases zur Überprüfung der korrekten Erzeugung der Produkte.

Listing 4.5: Test der Implementierung

```
1  #[cfg(test)]
2  mod test {
3      use super::{Director, ConcreteBuilder};
4
5      #[test]
6      fn basic() {
7          let mut concrete_builder = ConcreteBuilder::new();
8          let director = Director::new(&mut concrete_builder);
9          let concrete_product = director.get_product();
10         assert_eq!(2.0, concrete_product.cost);
11     }
12 }
```

Zunächst wird in Zeile 7 ein konkreter Builder (*ConcreteBuilder*) über die new-Methode erzeugt. Dieser Builder wird im Anschluss in Zeile 8 zur Erzeugung eines Directors genutzt. Der Director führt bei seiner Erzeugung über die Methode new die Schritte, die zum Bau des Produkts notwendig sind, aus. Zum Schluss wird das Produkt in Zeile 9 über die *get_product*-Methode zurückgegeben und in Zeile 10 wird überprüft, ob die Attribute richtig gesetzt wurden und äquivalent zu den zu erwartenden Attributen sind.

4.1.3 Builder Pattern nach Rust

Das Builder Pattern im vorherigen Teil wurde nach der Struktur implementiert, die die Gang of Four vorgibt [7, S. 121]. Eine andere Struktur findet sich in der eigenen Dokumentation von Rust [23, Kapitel Method Syntax]. Dabei wird der Director mit dem Builder verschmolzen und der Erzeugungsprozess eines Produkts liegt somit in einer Struktur. Dies stellt somit eine Vereinfachung des eigentlichen Builder Patterns dar. Eine beispielhafte Implementation findet sich anhand des Products aus dem vorherigen Teil.

Listing 4.6: Deklaration des Builders

```
1  struct ProductBuilder {
2      name: String,
3      id: u32,
```

```
4     cost: f32,
5 }
```

Es wird eine Struktur *ProductBuilder* deklariert. Diese besitzt die gleichen Attribute wie die Struktur, für die ein Builder erstellt wird (*name, id* und *cost* in diesem Fall). Darauf folgt die Implementation der Struktur, die Methoden zum setzen von Attributen anbietet. Zusätzlich könnten noch weitere Methoden zur Durchführung von Produktionsschritten angeboten werden.

Listing 4.7: Implementierung des Builders

```
1 impl ProductBuilder {
2     fn new() -> ProductBuilder {
3         ProductBuilder {name: "".to_owned(), id:  0, cost: 0.0}
4     }
5
6     fn name(&mut self, name: &str) -> &mut ProductBuilder {
7         self.name = name.to_owned();
8         self
9     }
10
11    fn id(&mut self, id: u32) -> &mut ProductBuilder {
12        self.id = id;
13        self
14    }
15
16    fn cost(&mut self, cost: f32) -> &mut ProductBuilder {
17        self.cost = cost;
18        self
19    }
20
21    fn build(&self) -> Product {
22        Product { name: self.name.to_owned(), id: self.id, cost:
              self.cost}
23    }
24 }
```

Zunächst gibt es eine Methode *new* zum Konstruieren einer Builder-Instanz. Diese verfügt über default-Werte, sodass keine inkonsistente beziehungsweise unvollständige Produkt-Instanz erstellt werden kann. Darauf folgen die Set-Methoden, um die Attribute zu setzen. Sie besitzen jeweils den Namen des jeweiligen Attributes (*name, id* und *cost* in diesem Fall).

Die Setter erwarten jeweils eine veränderbare Referenz (*&mut self* beziehungsweise *&mut ProductBuilder*), um die Attribute intern verändern zu können und den Wert, auf den das Attribut gesetzt werden soll. Den Rückgabewert bildet die veränderbare Referenz des Builders, um so über Method-Chaining eine einfache Objekterzeugung zu ermöglichen.

Darüber hinaus muss der Builder eine Methode zur letztendlichen Erzeugung des wirklichen Produkts anbieten (*build*). Diese erzeugt eine Produkt-Instanz mit den vorher gesetzten Attributen und liefert diese zurück. In dieser Implementation erwartet die Methode nur eine Referenz *&self*. Somit wird der Speicher, den der Builder belegt erst am Ende eines Scopes freigegeben. Eine andere Möglichkeit wäre, das Ownership in die build-Methode zu schieben (*self*). Somit würde am Ende der Methode der Speicher freigegeben werden. Allerdings wäre somit auch keine weitere Verwendung der Builder-Instanz möglich. Es muss je nach Anwendungsfall abgewogen werden, welche Möglichkeit passender ist.

Der implementierte Builder kann dann folgendermaßen zur Erzeugung von Produkten genutzt werden.

Listing 4.8: Verwendung des Builders

```
1  fn main() {
2      let product = ProductBuilder::new()
3                  .name("TestProduct")
4                  .id(1)
5                  .cost(1.5)
6                  .build();
7
8      println!("name: {}", product.name);
9      println!("id: {}", product.id);
10     println!("cost: {}", product.cost);
11 }
```

Zunächst wird in Zeile 2 der Builder über die new-Methode erzeugt. Danach können über Method-Chaining in Zeile 3 bis 5 die Attribute des Produkts gesetzt werden. Durch den abschließenden Aufruf der build-Methode in Zeile 6 wird das Produkt erzeugt und danach an die product-Variable gebunden. In den Zeilen 8 bis 10 werden zur Kontrolle die Attribute des Produkts auf der Standardausgabe ausgegeben.

Das Builder-Pattern aus der Rust Dokumentation lässt sich relativ simpel umsetzen und bietet einen einfachen Ansatz zur Erzeugung von Objekten.

4.1.4 Bewertung

Das Builder Pattern nach GoF kann analog in Rust umgesetzt werden. Dabei muss die Implementation um Lifetimes erweitert werden, da Trait Objects verwendet werden, die eine Referenz auf das Objekt hinter einem Trait darstellen. Somit kann der Compiler überprüfen, ob der Builder mindestens genau so lange lebt wie der Direktor. Daher kann es nicht zum use-after-free Fehler kommen, da der Direktor immer auf einen lebenden Builder bzw. validen Speicherbereich zugreift. [23, Kapitel Lifetimes]

Da diese Implementation Trait Objects benutzt, können darüber hinaus durch den dynamischen Dispatch auch zur Laufzeit die Builder ausgetauscht werden. Dies macht die Erzeugung von Objekten durch das Builder Pattern sehr flexibel. Allerdings führt dadurch der Methodenaufruf über eine vtable, wodurch die Geschwindigkeit des Programms im Vergleich zum statischen Dispatch verlangsamt wird.[23, Kapitel Trait Objects]

Darüber hinaus bietet es sich an, den Director um eine Methode zu erweitern, die die Konstruktion des Produkts durchführt. In diesem Beispiel wurde dies, aufgrund der Lesbarkeit und des einfach gehaltenen Beispiels, in die new-Methode verlagert, die zur Erzeugung des Directors genutzt wurde.

Da Rust keine Vererbung unter Strukturen unterstützt, muss sichergestellt sein, dass die Attribute der Produkt-Struktur und des Builders immer äquivalent sind. Andernfalls kommt es zu Inkonsistenzen oder Fehlern in der Erzeugung einer Struktur durch den Builder.

Das Builder Pattern aus der Rust Dokumentation wird vor allem auch deswegen genutzt, weil es kein Method-Overloading in Rust gibt. [23, Kapitel Method Syntax] Diese vereinfachte Version sollte genutzt werden, wenn die Konstruktion eines Objekts einfacher gestaltet ist. Bei einem komplexen Erzeugungsprozess bietet sich die Variante nach GoF an, da dieser eine bessere Lesbarkeit bietet, weil das Method-Chaining entfällt. Darüber hinaus bietet GoF auch die Möglichkeit zur Austauschbarkeit der Beschreibung der Erzeugung eines Objekts, also die Verwendung von untschiedlichen Buildern.

4.2 Abstract Factory

> Bietet eine Schnittstelle zu Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen ([7], S.107)

Die folgende Abbildung beschreibt den allgemeinen Aufbau einer Abstract Factory.

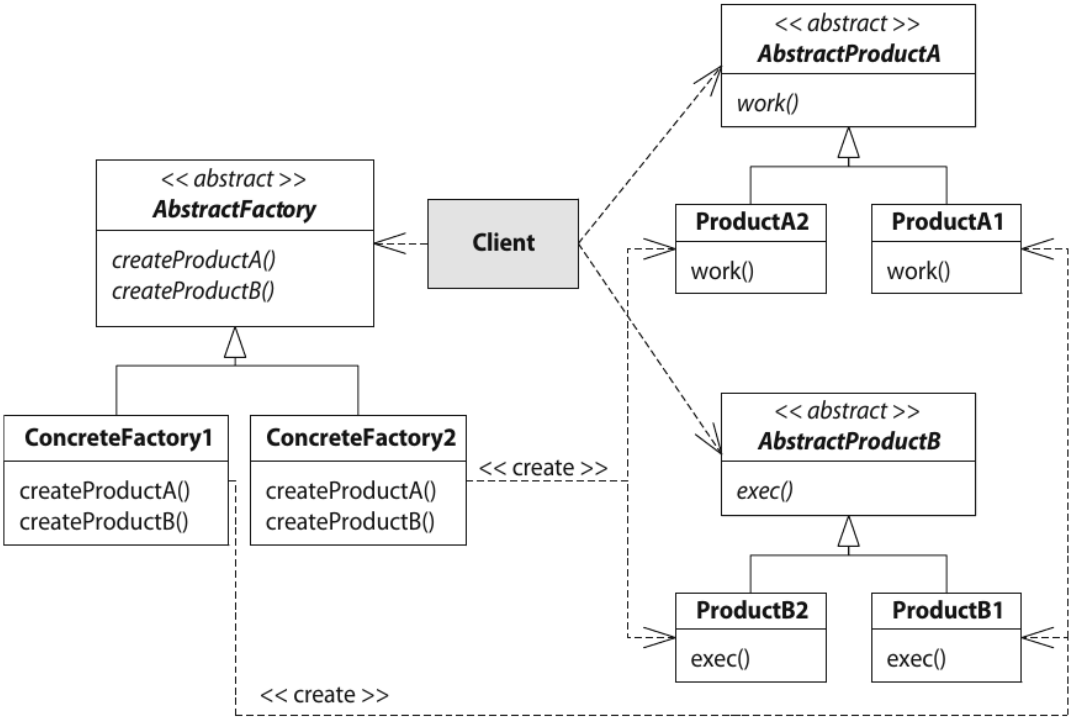

Abbildung 4.2: Abstract Factory nach GoF[7, S. 107ff], Bild aus [5, S.27]

Die Abstract Factory wird über die gleichnamige Schnittstelle nach außen repräsentiert. Sie bietet Methoden zur Erzeugung von konkreten Produkten an. Die Methoden zur Erzeugung von konkreten Produkten (*ConcreteProduct*) werden von konkreten Fabriken (*ConcreteFactory*) implementiert. Die Schnittstelle der Produkte wird durch das jeweilige *AbstractProduct* repräsentiert. Diese werden durch konkrete Produkte implementiert. Der Client greift nur über die Schnittstellen auf Fabrik und Produkte zu. [7, S.109f] Somit erfährt der Klient nichts über die interne Erstellung der Produkte.

[7] gibt drei beispielhafte Möglichkeiten zur Implementation an. Zum einen eine Fabrik, die das Singleton-Entwurfsmuster[1] nutzt, eine prototypbasierte und eine vererbungsbasierte Fabrik. ([7], S.111ff)

Im Folgenden wird eine einfache abstrakte Fabrik mit konkreten Fabriken gebaut.

[1] Siehe Singleton nach [7], S.157

4.2.1 Umsetzung einer einfachen Abstract Factory in Rust

Eine einfach in Rust umsetzbare Abstract Factory kann fast eins zu eins aus der der Abbildung 4.2 erstellt werden. Dabei werden die Schnittstellen mit Hilfe von Traits und die konkreten Fabriken mit Hilfe von Structs implementiert. In dem folgenden Beispiel stellen die Produktschnittstellen Methoden bereit, die zur Veranschaulichung des Anwendungsfalls und zum Testen genutzt werden.

Listing 4.9: Die Schnittstellen der Abstract Factory

```
trait AbstractFactory {
    fn create_product_a(&self) -> Box<ProductA>;
    fn create_product_b(&self) -> Box<ProductB>;
}

trait ProductA {
    fn work(&mut self);
    fn get_id(&self) -> String;
}

trait ProductB {
    fn exec(&mut self);
    fn get_id(&self) -> String;
}
```

Der Trait *AbstractFactory* bietet die Methoden *create_product_a* und *create_product_b* zur Erzeugung von *ProductA* und *ProductB* an. Die Methode gibt ein auf dem Heap liegendes Produkt zurück. Des Weiteren stellt jedes Produkte jeweils eine Schnittstelle *ProductA* und *ProductB* bereit, die produktspezifische Methoden bereithalten. Das *ProductA* bietet eine *work*- und eine *get_id*-Methode an. Das *ProductB* stellt die Methoden *exec* und *get_id* bereit. Die *get_id*-Methode wird für den Test der AbstractFactory genutzt.

4.2.2 Die konkreten Fabriken

Listing 4.10: Die Implementation der konkreten Fabriken

```
struct ConcreteFactory1;

impl AbstractFactory for ConcreteFactory1 {
    fn create_product_a(&self) -> Box<ProductA> {
        Box::new(ProductA1::new())
```

```
    }

    fn create_product_b(&self) -> Box<ProductB> {
        Box::new(ProductB1::new())
    }
}

struct ConcreteFactory2;

impl AbstractFactory for ConcreteFactory2 {
    fn create_product_a(&self) -> Box<ProductA> {
        Box::new(ProductA2::new())
    }

    fn create_product_b(&self) -> Box<ProductB> {
        Box::new(ProductB2::new())
    }
}
```

Die Implementation der *AbstractFactory*-Schnittstelle wird beispielhaft durch zwei konkrete Fabriken *ConcreteFactory1* und *ConcreteFactory2* verkörpert. Beide Structs besitzen keine Attribute[2], da in diesem Beispiel eine vereinfachte Implementation dargestellt wird. Man kann die Structs aber mit spezifischen Attributen ausstatten, auf die man bei der Erzeugung zurückgreifen kann, sofern man das möchte. Die konkreten Fabriken implementieren die *AbstractFactory*-Schnittstelle und stellen somit die Methoden zur Erzeugung der Produkte zur Verfügung. In der Implementation wird ein konkretes Produkt auf dem Heap erzeugt und zurückgegeben. Über die konkreten Fabriken steuert man also die Rückgabe des Produkts.

4.2.3 Die konkreten Produkte

Listing 4.11: Die Implementation der konkreten Produkte

```
struct ProductA1;

impl ProductA1 {
    fn new() -> Self {
        ProductA1
    }
}
```

[2] In Rust auch Unit-like Struct genannt (Siehe [23], Kapitel Structs)

```
8
9  impl ProductA for ProductA1 {
10     fn work(&mut self) {
11         println!("ProductA1");
12     }
13
14     fn get_id(&self) -> String {
15          "a1".to_string()
16     }
17 }
18
19 struct ProductA2;
20
21 impl ProductA2 {
22     fn new() -> Self {
23         ProductA2
24     }
25 }
26
27 impl ProductA for ProductA2 {
28     fn work(&mut self) {
29         println!("ProductA2");
30     }
31
32     fn get_id(&self) -> String {
33         "a2".to_string()
34     }
35 }
36
37 struct ProductB1;
38
39 impl ProductB1 {
40     fn new() -> Self {
41         ProductB1
42     }
43 }
44
45 impl ProductB for ProductB1 {
46     fn exec(&mut self) {
47         println!("ProductB1");
```

```
48      }
49
50      fn get_id(&self) -> String {
51          "b1".to_string().to_string()
52      }
53  }
54
55  struct ProductB2;
56
57  impl ProductB2 {
58      fn new() -> Self {
59          ProductB2
60      }
61  }
62
63  impl ProductB for ProductB2 {
64      fn exec(&mut self) {
65          println!("ProductB2");
66      }
67
68      fn get_id(&self) -> String {
69          "b2".to_string()
70      }
71  }
```

Die *ProductA*- und *ProductB*-Schnittstellen werden durch jeweils zwei konkrete Produkte implementiert. Die Structs implementieren die *work*- und *get_id*-Methode. Diese Methoden sind zum Test der Fabrik gedacht. In der *work*-Methode geben die Structs jeweils den Namen der eigenen Struct aus. Die *get_id*-Methode gibt einen String zurück, welcher der Name der jeweiligen Struct ist. Somit gibt jede Methode in diesem Beispiel eine individuelle ID zurück, die zum Testen genutzt wird. Des Weiteren bieten die Structs jeweils eine Methode *new* zur Instantiierung der Structs. Da die konkreten Produkte keine Attribute besitzen, werden bei der Instantiierung keine Werte übergeben.

4.2.4 Test der Implementierung

Es folgt die Implementierung eines Testcases zur Darstellung der Verwendung der Fabrik und zur Überprüfung der korrekten Erzeugung der Produkte.

Listing 4.12: Der Test der Implementierung

```
1  #[cfg(test)]
```

```
2 mod test {
3     use super::{AbstractFactory, ConcreteFactory1, ConcreteFactory2
          };
4
5     #[test]
6     fn basic() {
7         let abstract_factory = ConcreteFactory1;
8
9         let a1 = abstract_factory.create_product_a();
10        assert_eq!("a1".to_string(), a1.get_id());
11
12        let b1 = abstract_factory.create_product_b();
13        assert_eq!("b1".to_string(), b1.get_id());
14
15
16        let abstract_factory = ConcreteFactory2;
17
18        let a2 = abstract_factory.create_product_a();
19        assert_eq!("a2".to_string(), a2.get_id());
20
21        let b2 = abstract_factory.create_product_b();
22        assert_eq!("b2".to_string(), b2.get_id());
23    }
24 }
```

In Zeile 7 wird zunächst eine konkrete Fabrik *ConcreteFactory1* instantiiert und an die Variable *abstract_factory* gebunden. In den Zeilen 9-22 werden durch Aufruf der *create_product* Produkte erzeugt und mit ihrer zu erwartenden ID verglichen.

4.2.5 Conditional Compilation

Bei der Kompilierung des Quellcodes können Flags angegeben werden, die zur bedingten Kompilierung genutzt werden können. Dabei muss der Compiler *rustc* mit der Option *–cfg FLAG* ausgeführt werden.[23, Kapitel Conditional Compilation]

Das folgende Beispiel veranschaulicht die Verarbeitung eines übergeben Flags.

Listing 4.13: Beispiel zur bedingten Kompilierung

```
1 #[cfg(test)]
2 mod test {
```

```
3     use super::{AbstractFactory, ConcreteFactory1, ConcreteFactory2
          };
4     #[test]
5     fn test_conditional_compilation() {
6         let abstract_factory: Box<AbstractFactory> = if cfg!(cf1) {
7             Box::new(ConcreteFactory1)
8         } else {
9             Box::new(ConcreteFactory2)
10        };
11
12        let a = abstract_factory.create_product_a();
13        let b = abstract_factory.create_product_b();
14
15        if cfg!(cf1) {
16            assert_eq!("a1".to_string(), a.get_id());
17            assert_eq!("b1".to_string(), b.get_id());
18        } else {
19            assert_eq!("a2".to_string(), a.get_id());
20            assert_eq!("b2".to_string(), b.get_id());
21        }
22    }
23 }
```

Zunächst wird in Zeile 6 bis 10 die auf dem Heap liegende abstrakte Fabrik erzeugt (*Box<AbstractFactory>*). Dabei wird durch eine if-Konstruktion die konkrete Fabrik ausgewählt. Es wird durch das *cfg!*-Makro[3] geprüft, ob dem Compiler das Flag *cf1* übergeben wurde. Falls das Flag übergeben wurde, so wird ConcreteFactory1 als konkrete Fabrik genutzt, und sonst ConcreteFactory2.

In Zeile 12 und 13 werden zwei Produkte a und b über die Methoden der Fabrik erzeugt. Anschließend wird wieder über das *cfg!*-Makro geprüft, ob das Flag *cf1* übergeben wurde und in Abhängigkeit davon die zugehörige Tests durchgeführt.

4.2.6 Bewertung

Die Beschreibung der Abstract Factory nach GoF konnte sehr gut und ohne komplizierte Umwege in Rust überführt werden.

Rust bietet keine Vererbung innerhalb von Strukturen an. Somit kann es zu Redundanzen in der Implementation kommen. Es kann zum Beispiel keine gemeinsame Funktionalität von

[3] https://doc.rust-lang.org/std/macro.cfg!.html

Produkten in eine höher liegende Struktur geschoben werden. Es gibt dazu Ansätze über Referenzen in einer Struktur auf andere Strukturen eine Vererbungshierarchie aufzubauen. Dabei werden die Methodenaufrufe an die passende Struktur weitergegeben. Allerdings sind diese Ansätze Workarounds und man muss abwägen, ob solche Konstrukte eine passende Lösung sind. Rust forciert das Composition-over-Inheritance-Prinzip und daher sollte die Problemlösung auf diesem Weg durchgeführt werden.

Durch die Verwendung von Trait Objects (Zeiger auf ein auf dem Heap liegendes Produkt, Box<Product>) wird dynamischer Dispatch durchgeführt. Somit arbeitet das Programm über eine vtable und ist durch die Indirektion der Methodenaufrufe langsamer im Vergleich zum statischen Dispatch. [23, Kapitel Trait Objects]

Durch die Möglichkeit der bedingten Kompilierung, kann eine einfache Konfiguration von außen vorgenommen werden, ohne die eigentliche Implementation anzupassen. Somit kann z.B. je nach System eine andere Fabrik zur Produkterzeugung genutzt werden.

4.3 Factory Pattern

Im Folgenden wird eine Variante des Factory Pattern implementiert, die auf Makros aufbaut. Die Fabrik soll die Möglichkeit bieten, beliebige Produkte, die einen bestimmten Trait implementieren, zu erstellen. Aufgrund der bisher, im Vergleich zu C++, schwach vorhandenen Polymorphie, kann der Typ der Objekte zur Laufzeit nicht ermittelt werden, da bei der Kompilierung die Typ Informationen gelöscht werden.[23, Kapitel Trait Objects] Das Makro soll bei der Parameterübergabe einen Trait und eine beliebige Anzahl von Beschreibungen zur Erstellung von Produkten enthalten (`create_factory!(Produkt, NameProdukt1 => Produkt1, NameProdukt2 => Produkt2`). Die Produkte können über eine parametrierbare Methode erzeugt werden. Diese Variante ähnelt einer Factory in C++ mit Switch-Case-Anweisungen und stellt somit einen sehr vereinfachten Ansatz dar.

4.3.1 Definition des Makros

Zunächst befassen wir uns mit der Definition des Makros *create_factory!*. Der Übersichtlichkeit wegen, wird das Makro aufgeteilt.

Listing 4.14: Pattern Matching des Makros

```
1 macro_rules! create_factory (
2     ($base:ty, $($k:pat => $v:expr),+) => {
```

Zunächst findet sich der Teil des Makros, der sich mit dem Matching befasst. Die Regel gibt an, dass an erster Stelle ein Typ erwartet wird, der an die Variable *$base* gebunden wird. Dies ist der Typ des Produkts, der durch die Factory erzeugt werden kann. Danach folgen mit Kommata getrennt ein oder mehr Produkte, die durch die Fabrik verfügbar gemacht werden sollen. Diese werden in dem Format *Bezeichner => Erzeugung des Produkts* angegeben. Der Bezeichner wird an die Variable *$k* und das Produkt an die Variable *$v* gebunden. *$k* ist vom Typ *pat*, da diese Variable in einem match-Statement[23, Kapitel Match] zum Pattern Matching genutzt wird. Die Variable *$v* ist vom Typ expr, da dies die Anweisung zur Erstellung eines Produkts in der Fabrik verkörpert.

Wird das Makro, in der im Pattern vorgegebenen Art, aufgerufen, so wird folgender Code durch die Expansion des Makros eingefügt.

Listing 4.15: Expansion des Makros

```
1         {
2             struct Factory;
3
4             impl Factory {
5                 fn get(&self, instance_id: &str) -> Option<$base> {
6                     match instance_id {
7                         $(stringify!($k) => Some(Box::new($v)),)*
8                         _ => {
9                             None
10                        }
11                    }
12                }
13
14                fn new() -> Self {
15                    Factory
16                }
17            }
18
19            Factory::new()
20        }
21    };
22);
```

Die Fabrik besitzt in diesem Beispiel keine eigenen Attribute und wird daher in Zeile 2 als leere Struktur deklariert. In Zeile 4 bis 17 folgt die Implementation der Struktur. Die Fabrik bietet die zwei Methoden *get* und *new* an. Die *get*-Methode erwartet einen String, der den

Bezeichner des Produkts darstellt, und liefert ein in Option gekapseltes Objekt, das den bei der Verwendung des Makros angegebenen Trait implementiert. Die Funktion besteht nur aus einem match-Statement, dessen Größe abhängig von der Anzahl der übergebenen Produkte bei Aufruf des Makros ist. Das match-Statement matcht die Variable *instance_id*. Die *Repetition* in Zeile 7 expandiert zu den Pfaden des match-Statements, die aus den verfügbaren Instanzen bestehen. In jeden Pfad wird die jeweilige *Expression* eingefügt, die bei Aufruf des Makros übergeben wird. Somit wird ein match-Statement erzeugt, welches in jedem Pfad eines Produkts jeweils die Beschreibung zur Erzeugung des jeweiligen Produkts enthält. Die Methode *new* gibt lediglich eine Instanz der Fabrik zurück. Möchte man bei der Erstellung der abstrakten Fabrik noch andere Funktionalität einbauen, so kann man dies zentral in der Methode *new* durchführen. Am Ende folgen noch die schließenden Klammern. Das Resultat ist eine einfach verwendbare und flexible Fabrik, die mit Hilfe eines *Makros* und *match*-Statements konstruiert wird.

4.3.2 Die Testumgebung

Da die Fabrik ein wenig anders aufgebaut ist, wird eine neue Testumgebung implementiert. Sie ähnelt dem Beispiel aus der einfachen abstrakten Fabrik. Daher wird das Beispiel nur kurz beschrieben.

Listing 4.16: Die Testumgebung für das Factory Pattern

```
1  trait Product {
2      fn get_costs(&self) -> u32;
3  }
4
5  struct Product1 {
6      length: u32,
7      width: u32,
8  }
9
10 impl  Product1 {
11     fn new(length: u32, width: u32) -> Self {
12         Product1 {length: length, width: width}
13     }
14 }
15
16 impl Product for Product1 {
17     fn get_costs(&self) -> u32 {
```

```
18          self.length * self.width
19      }
20  }
21
22  struct Product2 {
23      length: u32,
24  }
25
26  impl Product for Product2 {
27      fn get_costs(&self) -> u32 {
28          (self.length * self.length)
29      }
30  }
31
32  impl Product2 {
33      fn new(length: u32) -> Self {
34          Product2 {length: length}
35      }
36  }
```

Es gibt eine Produkt-Schnittstelle (Zeile 1 bis 3), die die Methode *get_costs* anbietet. Die Methode liefert einen unsigned, 32-Bit Integer zurück. Die Schnittstelle wird von den Produkten *Product1* und *Product2* implementiert. Beide Structs besitzen jeweils beispielhaft Attribute und unterschiedliche Implementationen zur Berechnung der Kosten (*get_costs*). Darüber hinaus besitzen beide Produkte eine Methode *new*, um sie jeweils zu instantiieren.

4.3.3 Der Test des Makros

Die Produkte aus dem vorherigen Abschnitt werden für die folgenden Testfälle genutzt.

Listing 4.17: Test des Makros

```
1  #[cfg(test)]
2  mod test {
3      use super::{Product, Product1, Product2};
4
5      #[test]
6      fn test_macro() {
7          let factory = create_factory!(Box<Product>, Product1 =>
               Product1::new(2, 2), Product2 => Product2::new(3));
8          let p1 = factory.get("Product1").unwrap();
```

```
9           let p2 = factory.get("Product2").unwrap();
10
11          assert_eq!(4, p1.get_costs());
12          assert_eq!(9, p2.get_costs());
13      }
14  }
```

In Zeile 7 wird mit Hilfe des Makros *create_factory!* die Fabrik erstellt und an die Variable *factory* gebunden. Als erstes Argument wird der erwartete Rückgabewert der Instanzen der Fabrik angegeben. In diesem Fall möchten wir ein auf dem Heap liegendes Produkt (*Box<Product>*) zurückerhalten. Die weiteren Argumente sind die Produkte, die durch die Fabrik angeboten werden sollen (*Product1* und *Product2*). In Zeile 8 und 9 wird jeweils ein *Product1* und *Product2* an Variablen gebunden, indem sie mit dem passenden Bezeichner über die Fabrik und der Methode *get* erstellt werden. Die gekapselten Produkte werden durch Aufruf der *unwrap* Methode aus dem Option Typ bewegt. Darauf folgend wird in Zeile 11 und 12 getestet, ob die richtigen Produkte durch die Fabrik erzeugt wurden. Dafür wird durch den Aufruf der *get_costs*-Methode geprüft, ob die zurückgegebenen Kosten gleich den zu erwartenden Kosten sind.

Darüber hinaus wurde ein Negativtest implementiert, der den Fall eines nicht in der Fabrik vorhandenen Produkts testen soll.

Listing 4.18: Negativtest

```
1  #[cfg(test)]
2  mod test {
3      use super::{Product, Product1, Product2};
4
5      #[test]
6      #[should_panic]
7      fn test_not_existing_product() {
8          let factory = create_factory!(Box<Product>, Product2 =>
               Product2::new(2), Product1 => Product1::new(2, 2));
9          factory.get("NotExisting").unwrap();
10     }
11 }
```

Dass ein Negativtest dargestellt ist, erkennt man anhand der Annotation *should_panic*. Diese sagt aus, dass eine Methode zum Absturz des Programms führen soll. In Zeile 8 wird wie auch schon im vorherigen Test eine Fabrik mittels des Makros konstruiert und an die Variable *factory* gebunden. In der nächsten Zeile wird dann bei dem Aufruf der *get_text* Methode ein

Bezeichner übergeben, unter dem kein Objekt hinterlegt ist. Daher wird *None* von der Methode zurückgegeben und bei Aufruf der Methode *unwrap* wird der Prozess fehlerhaft beendet.

4.3.4 Bewertung

Dieser Ansatz zur Umsetzung der Fabrik geht davon aus, dass die zu erzeugenden Produkte zur Kompilezeit feststehen. Zur Laufzeit können keine weiteren Objekte zur Fabrik hinzugefügt werden. Dies hat den Vorteil der Kontrolle über die Größe der Fabrik, da diese bereits zur Kompilezeit feststeht. In Systemen, in denen die Speicherkapazität von wichtiger Bedeutung, ist kann dieser statische Ansatz von Vorteil sein.

Da bei der Benutzung des Makros ein beliebiger Ausdruck übergeben werden kann, der zur Erzeugung der Objekte genutzt wird, kann auch ein Block von Anweisungen übergeben werden. Sofern noch mehr Schritte zur Erzeugung eines Objekts notwendig sind oder andere Anweisungen ausgeführt werden sollen, stellt dies eine praktische Möglichkeit dar. Dieser Ansatz lässt eine große Offenheit zur Kontrolle der Erzeugung von Objekten.

Zur Zeit ist es mit Makros nicht möglich Funktionen zu konstruieren, deren Namen den Inhalt einer Variable enthalten (z.B. fn create_$VARIABLE). Durch die Überarbeitung des Makrosystems könnte dies bald möglich sein.[2] Dadurch wäre die Erstellung von Methoden über ein Makro möglich, die ein spezifisches Produkt erzeugen (z.B. create_product_a).

Darüber hinaus kann der Compiler die Erzeugung eines Objekts durch die Fabrik, also das match-Statement, optimieren, wenn bereits zur Kompilezeit feststeht, dass ein bestimmtes Objekt erzeugt werden soll. Dabei wird der Ausdruck zur Erzeugung des Objekts in die passende Stelle des Quellcodes eingefügt. Somit wird ein Objekt in konstanter Zeit erzeugt. Ist erst zur Laufzeit bekannt, welches Objekt erzeugt werden soll, so ist die Erzeugung von der Größe des erzeugten match-Statements abhängig, also von der Anzahl der Produkte, die die Fabrik erzeugen kann.

Um eine dynamische Lösung zu gestalten, die auch zur Laufzeit um erzeugbare Produkte erweitert werden kann, muss ein virtueller Konstruktor aufgerufen werden, da die Typ-Informationen beim dynamischen Dispatch gelöscht werden[23, Kapitel Trait Objects]. Ein Ansatz wäre dabei, dass die Produkte, die erzeugt werden sollen, den Trait Clone implementieren. Dadurch kann die Fabrik durch Aufruf der *clone*-Methode ein Produkt erzeugen. Die Fabrik besitzt eine Map der erzeugbaren Produkte, welche über eine Methode hinzugefügt bzw. geklont werden können. Eine beispielhafte Implementation findet sich auf der beiliegenden CD.

5 Finite-State Machine

Ein endlicher Automat (engl. Finite-State Machine, kurz FSM) kann zur Lösung vieler Probleme beitragen. Beispiele für die Verwendung sind unter anderem die Repräsentation von echten elektronischen Automaten (Zugfahrkarte, Parkautomat und so weiter), reaktiven Systemen oder auch das Parsen von Protokollen (TCP, HTTP et cetera) und natürlichen Sprachen.

Endliche Automaten werden durch ein Quintupel aus Eingabealphabet, Menge von Zuständen, Startzustand, Übergangsfunktion und Menge von akzeptierenden Zustände dargestellt. [10, S.54]

5.1 Erster Ansatz

Die Implementation in Rust soll an die formale Beschreibung von endlichen Automaten gelehnt sein (vgl. [10], S. 45ff). Einen wichtigen Bestandteil der Implementation bildet die Übergangsfunktion. Eine Möglichkeit um die Übergangsfunktion zu repräsentieren ist zum Beispiel ein zweidimensionales Array. Es verkörpert eins zu eins die Übergangstabelle für den endlichen Automaten. Darüber hinaus beträgt die Zeit zur Ermittlung des Nachfolgezustands aus aktuellem Zustand und Eingabezeichen O(1). Eine weitere Möglichkeit, um die Übergangstabellen darzustellen, wären Listen. Allerdings beträgt dann die Zugriffszeit auf den nachfolgen Zustand O(n).[1] Eine andere Möglichkeit stellt die Benutzung einer Hash Map dar. Diese haben eine Zugriffszeit von O(1) und des Weiteren besteht die Möglichkeit zum dynamischen Hinzufügen von Status, sofern dies der Anwendungsfall verlangt. Die Zugriffszeit auf die Elemente der Hash Map hängt von der genutzten Hash-Funktion und den gehashten Daten ab. Darüber hinaus wurde in der Implementation eine Hash Map benutzt, da somit der Zugriff durch das später erstellte Makro einfacher gestaltet ist.

5.1.1 Die Implementation des Automaten

Im Folgenden wird eine beispielhafte Implementation eines endlichen Automaten beschrieben. Der Automat wird dabei in der Form eines Moore-Automaten implementiert. Das heißt, eine

[1] Angenommen die Liste ist als verkettete Liste implementiert.

Aktion wird dem Status zugeordnet und nicht einer Transition, wie das bei Mealy-Automaten der Fall ist.[17, S.28] Dabei wurde der Quellcode der Übersicht wegen aufgeteilt. In der Implementation des endlichen Automaten wird vorausgesetzt, das jeder Status einen individuellen Namen hat.

Listing 5.1: Use-Statements

```
use std::collections::HashMap;
use std::hash::Hash;
```

Zuerst werden die für die Hash Map benötigten Structs und Traits über die use-Anweisung eingebunden bzw. deren Verwendung vereinfacht. Es wird die HashMap-Struktur selber und darüber hinaus noch der Trait *Hash* inkludiert. Der Hash-Trait wird zur Generierung von Hash-Implementation über das *derive*-Attribut benötigt.

Anschließend kann die Struktur des endlichen Automaten implementiert werden.

Listing 5.2: Der endliche Automat

```
struct StateMachine<Token: Eq + Hash> {
    initial_state: String,
    current_state: Option<String>,
    accepted: bool,
    states: HashMap<String, State>,
    transitions: HashMap<(String, Token), String>,
}
```

Der endliche Automat besitzt als Attribut den Namen des Startzustands (*initial_state*), den Namen des aktuellen Zustands (*current_state*) und ein Flag, dass signalisiert, ob der Automat in einem akzeptierenden Zustand gelandet ist (*accepted*). Der Name des aktuellen Zustands ist in Option gekapselt. Ist der aktuelle Zustand None, so ist der Automat noch nicht gestartet worden bzw. in einem akzeptierenden Zustand. Dies kann man anhand des accepted-Flags feststellen. Zusätzlich dazu besitzt der Automat eine Hash Map *states*, die den Namen eines States auf das dazugehörige State-Objekt mappt. Darüber hinaus enthält der Automat eine Hash Map, die die Übergangstabelle repräsentiert (*transitions*). Als Schlüssel der Map dient ein Tupel, welches aus Status und Eingabezeichen (*Token*) besteht. Der Wert, auf den der Schlüssel gemappt wird, ist wiederum auch ein Status. Im Grunde genommen wird dadurch die Übergangsfunktion $f : Status x Eingabezeichen \rightarrow Status$ verkörpert. Das Eingabezeichen (*Token*) muss die Traits *Eq* und *Hash* implementieren, damit es als Schlüssel in der Map genutzt werden kann. Dabei wird Hash zum hashen des Schlüssels benötigt und Eq, um im Kollisionsfall den äquivalente Schlüssel zu finden.

Danach wird die Funktionalität, die der Automat anbietet, implementiert.

Listing 5.3: Die Implementation der Methoden des endlichen Automaten

```
impl<Token: Eq + Hash> StateMachine<Token> {
    fn new(initial_state: String, states: HashMap<String, State>,
        transitions: HashMap<(String, Token), String>) -> Self {
            StateMachine {initial_state: initial_state.to_owned(),
                current_state: None, states: states, transitions:
                transitions, accepted: false}
    }

    fn start(&mut self) {
        self.current_state = Some(self.initial_state.clone());
        self.execute_current_state();
    }

    fn execute_current_state(&self) {
        self.states.get(&self.current_state.clone().unwrap()).
            unwrap().act();
    }

    fn add_state(&mut self, state_to_add: State) {
        let state_name = state_to_add.name.clone();
        self.states.insert(state_name, state_to_add);
    }

    fn add_transtion(&mut self, stateinp: (String, Token),
        following_state: String) {
        self.transitions.insert(stateinp, following_state).unwrap()
            ;
    }

    fn next(&mut self, input: Token) {
        let current_state_name = self.current_state.clone().unwrap
            ();

        match self.transitions.get(&(current_state_name, input)){
            Some(following_state) => self.current_state = Some(
                following_state.to_owned()),
            None => println!("No_transition"),
```

```
30          };
31
32          let current_state = self.states.get(&self.current_state.
                clone().unwrap());
33
34          current_state.unwrap().act();
35          if current_state.unwrap().accepting  {
36              self.current_state = None;
37              self.accepted = true;
38          }
39      }
40  }
```

Die Implementation des endlichen Automaten umfasst verschiedene Methoden. Zunächst befindet sich in Zeile 1 bis 4 die *new*-Methode. Diese erwartet einen Anfangszustand, eine Hash Map, in der sich die Zustände befinden und eine Hash Map, die die Transitionen enthält. Durch den Aufruf der *new*-Methode der endliche Automat aus den gegebenen Parametern konstruiert und zurückgegeben. Alternativ können auch leere Hash Maps übergeben werden und die Zustände und Transitionen über Methoden, die im Folgenden besprochen werden, hinzugefügt werden.

Die *start*-Methode versetzt den Automaten in einen ausführbaren Zustand und führt die Aktion des Startzustands aus. Dadurch ist Erzeugung des Automaten möglich, ohne direkt eine Ausführung des Automaten zu starten. Die *start*-Methode muss immer vor Benutzung des Automaten ausgeführt werden. Des Weiteren gibt es eine *execute_current_state*-Methode, um die Aktion des aktuellen Zustands auszuführen. Die Aktion des aktuellen Zustands wird nach jeder Transition automatisch durch den Automaten aufgerufen und muss daher nicht manuell durch die Methode execute_current_state aufgerufen werden.

Darüber hinaus besitzt der endliche Automat eine *add_state*- (Zeile 15 bis 18) und eine *add_transition*-Methode (Zeile 20 bis 22). Dadurch kann der Automat auch nach der Erstellung oder auch zum Beispiel zur Laufzeit dynamisch erweitert werden. Die add_state-Methode erwartet einen Status, der zur Hash Map *states* hinzugefügt wird. Die add_transition-Methode erwartet ein Tupel aus Name eines Zustands und Eingabezeichen, und Name eines Folgezustands. Also die Beschreibung der Transition von einem Zustand über ein Eingabezeichen in einen nachfolgenden Zustand

Abschließend bietet der endliche Automat eine *next*-Methode (Zeile 24 bis 39) zum Zustandswechsel an. Diese erwartet ein Eingabezeichen. Zunächst wird in Zeile 25 der Name des aktuellen Zustands geklont und an die Variable *current_state_name* gebunden. Der String wird geklont, da der get-Methode eine Referenz auf den Namen übergeben wird. Somit könnte der aktuelle Zustand ohne Verwendung eines Klons nicht verändert werden, da die get-Methode lesenden Zugriff auf die Referenz hätte. Dies würde gegen das Ownership-Modell verstoßen.

In Zeile 27 bis 30 wird die eigentliche Transition durchgeführt. Anhand des Tupels aus aktuellem Status und dem erhaltenen Eingabezeichen wird der Name des nachfolgenden Zustands aus der Map geholt. Dieser wird der neue aktuelle Zustand. Findet sich in der Hash Map kein nachfolgender Zustand, so wird *None* von der get-Methode der HashMap zurückgeliefert und daher wird über das println!-Makro der Text 'No transition' auf der Standardausgabe ausgegeben. Der aktuelle Zustand wird dabei nicht neu gesetzt.

Anschließend wird in Zeile 32 mit dem Namen des neuen aktuellen Zustands die get-Methode der Hash Map states aufgerufen. Diese liefert in Option gekapselt eine Referenz auf das eigentliche Statusobjekt zurück, sofern dieses vorhanden ist. Daher wird in Zeile 34 die Methode unwrap der Option Struktur aufgerufen, um die Referenz aus der Kapselung zu bewegen und die Methode act aufgerufen, um die Aktion des aktuellen Status auszuführen.

Am Ende der Methode wird überprüft, ob der aktuelle Zustand ein akzeptierender Zustand ist. Wenn dies zutrifft, so wird das accepted-Flag des Automaten gesetzt und der aktuelle Zustand wird auf None gesetzt. Die Ausführung des Automaten wäre somit beendet.

5.1.2 Die Implementation der Zustände und Aktionen

Es folgt die Implementation der Zustands-Struktur, die von dem endlichen Automaten benutzt wird.

Listing 5.4: Die Zustands-Struktur

```
1 struct State {
2     accepting: bool,
3     name: String,
4     action: Option<Action>
5 }
```

Ein Status verfügt über das *accepting*-Flag, welches signalisiert, ob der Status ein akzeptierender Status ist. Darauf folgt der Name, der in dieser Implementation für jeden Status in einem endlichen Automaten individuell sein muss. Darüber hinaus verfügt diese Implementation eines endlichen Automat über die Möglichkeit einem Status ein sogenanntes *Action*-Objekt

zu übergeben. Dieses ermöglicht das Hinzufügen einer spezifischen und selber zu implementierenden Aktion, die zum Beispiel beim Betreten des Status ausgeführt werden kann. Eine Beispielhafte Verwendung findet sich im Test der Implementierung.

Listing 5.5: Die Methoden des Zustands

```
impl State {
    fn new(accepting: bool, name: &str, action: Option<Action>) ->
       Self {

        State {
                accepting: accepting,
                name: name.to_owned(),
                action: action
        }
    }

    fn act(&self) {
        match self.action {
            Some(ref action) => action.act(),
            None => (),
        }
    }
}
```

Die State-Struktur bietet zwei Methoden an. Zum einen eine *new*-Methode zum Erzeugen eines neuen Zustands. Diese konstruiert aus den gegeben Parametern einen neuen Zustand.

Darüber hinaus bietet ein Status noch eine Methode *act* an, die die Aktion eines Status ausführt, sofern eine vorhanden ist. Das Vorhandensein wird über das match-Statement in Zeile 12-15 überprüft. Dabei wird in Zeile 13 das Schlüsselwort *ref* benutzt. Die Methode act erhält eine Referenz auf self (&self), also eine Referenz auf das Objekt, auf dem die act-Methode ausgeführt wird (&self). Daher muss man beim match-Statement angeben, dass man eine Referenz auf das in &self liegende action-Attribut erhalten möchte.

Listing 5.6: Die Implementation der Aktion

```
struct Action {
    action: Box<Fn()>,
}

impl Action {
```

```
6       fn new(action: Box<Fn()>) -> Self {
7           Action {action: action}
8       }
9
10      fn act(&self) {
11          let act = &(self.action);
12          act();
13      }
14  }
```

Das Action-Objekt besitzt eine auf dem Heap liegende Closure (Box<Fn()>) (Siehe [23, Kapitel Closures]). Die Action-Struktur bietet eine statische Methode zum erzeugen eines neuen Action-Objekts aus einer gegebenen Closure an. Des Weiteren stellt diese eine *act*-Methode zum Ausführen der Closure zur Verfügung. Die Verwendung von Closures wird in der Implementation des Tests veranschaulicht.

5.2 Test des ersten Ansatzes

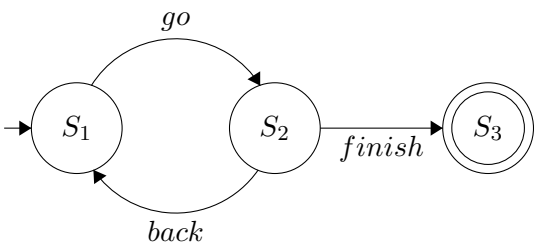

Abbildung 5.1: Automat zum Testen des Automaten

Im Folgenden wird ein Test des ersten Ansatzes durchgeführt, um die korrekte Implementation und darüber hinaus eine beispielhafte Verwendung darzustellen. Der Test nimmt den obigen endlichen Automaten aus Abbildung 5.1 als Grundlage. Zur besseren Übersicht wird der Test in seine Einzelteile aufgespalten. Aktionen, die bei Betreten eines Zustands ausgeführt werden, sind nicht dargestellt.

Für den Test wird ein Kontext implementiert, der durch den endlichen Automaten verändert werden soll.

Listing 5.7: Deklaration des Kontextes

```
1   struct TestContext {
2       name: String,
```

```
3         data: u32,
4     }
```

Diese Struktur besitzt die Attribute *name* vom Typ String und *data* vom Typ u32. Durch die Verwendung eines Kontext-Objekts, soll eine Möglichkeit gezeigt werden, wie man trotz nicht vorhandenem Kontext in der Implementation des endlichen Automaten (vgl. [7], S. 400), diesen selber einführen kann.

Dieser Kontext wird von der folgenden Testfunktion *basic_implementation_test* zum Testen des endlichen Automaten verwendet. Dabei soll der Kontext durch einen oder mehrere Status verändert werden können.

Listing 5.8: Instanziierung des Kontextes

```
1     #[test]
2     fn basic_implementation_test() {
3         let context = Rc::new(RefCell::new(TestContext {name: "
              Startname".to_owned(), data: 0}));
```

In Zeile 3 wird das Kontext-Objekt erstellt und an die Variable *context* gebunden. Bei der Erstellung werden zwei weitere Datentypen aus Rust benötigt. Das Kontext-Objekt soll lesend und schreibend von mehreren anderen Objekten benutzt werden können. Dies wird zunächst durch die Prinzipien des *Borrowing*[23, Kapitel References and Borrowing] verhindert, da wir durch dieses Prinzip entweder viele Leser oder einen Schreiber (und damit gleichzeitig Leser) haben können. Daher werden hier zwei neue Typen benötigt. Einer dieser Typen ist *RefCell*. Dieser Datentyp ermöglicht die Einführung von Speicherbereichen, die zur Laufzeit dynamisch nach den Regeln des *Borrowing* überprüft werden (vgl. [26]). Dies wird benötigt, da der Kontext durch mehrere Zustände veränderbar sein soll. Der Kontext wird in einem solchen *RefCell*-Typ gelagert. Darüber hinaus muss das Objekt an mehrere Besitzer verteilt werden. Dies verstößt zunächst gegen das Ownership-Modell, welches aussagt, dass ein Objekt immer nur genau einen Besitzer haben kann. Damit dieses Objekt mehrere Besitzer haben kann, wird der Typ *Rc* verwendet. Die Abkürzung *Rc* steht für *reference-counted* und stellt eine andere Art von Zeiger dar, die ähnlich wie der *shared_ptr* aus C++ (vgl. [1]) auf Referenzzählung basiert. Diese Art von Zeiger bietet die Methode *clone* an, die den internen Zähler der vorhandenen Referenzen hochzählt. Sobald ein "Klon"dieses Objekts freigegeben wird, wird der interne Zähler dekrementiert. Sobald der interne Zähler bei 0 angekommen ist, wird der Speicherbereich freigegeben, da niemand mehr eine Referenz auf das Objekt hat (vgl. [28]).

Somit kann der Kontext nun durch die Kapselung *Rc<RefCell<TestContext»*, ohne Verlust der Sicherheit, an mehrere Zustände verteilt werden

Listing 5.9: Definition der Aktionen

```
1         let cloned_context = context.clone();
2         let s2_action = Action::new(Box::new(move || {
              cloned_context.borrow_mut().data = 21; cloned_context.
              borrow_mut().name = "TestName".to_owned()}));
3         let cloned_context = context.clone();
4         let s3_action = Action::new(Box::new(move || {println!("
              Final␣State:␣Data:␣{},␣Name:␣{}", cloned_context.borrow
              ().data, cloned_context.borrow().name)}));
```

Zunächst wird die *clone*-Methode der Rc-Struktur aufgerufen und der Zeiger auf die Kontext-Instanz an die Variable *cloned_context* gebunden. Dadurch wird der Referenzzähler inkrementiert. In Zeile 2 wird eine Action-Instanz erstellt und dabei eine Closure (siehe [23, Kapitel Closures]) übergeben. Die Closure liegt auf dem Heap (*Box::new*, siehe [25]). Durch das vorangehende Schlüsselwort *move* wird sichergestellt, dass die Closure das Ownership für die in der Closure verwendeten Objekte übernimmt. Somit wird der Besitz von *cloned_content* der Closure übertragen. Die Closure setzt die Attribute des Kontextes auf neue Werte, um den korrekten Aufruf der Aktion zu überprüfen. In Zeile 3 bis 4 wird wieder zunächst eine Referenz auf den Kontext erzeugt und der Besitz an die Closure der Aktion *Action2* übergeben. Die Closure dieser Aktion gibt die Attribute des Kontextes auf der Standardausgabe aus.

Die Aktionen werden für die Definition der Zustände benutzt.

Listing 5.10: Definition der Zustände

```
1         let s1 = State::new(false, "s1", None);
2         let s2 = State::new(false, "s2", Some(s2_action));
3         let s3 = State::new(true, "s3", Some(s3_action));
4
5         let mut states = HashMap::new();
6         states.insert(s1.name.to_owned(), s1);
7         states.insert(s2.name.to_owned(), s2);
8         states.insert(s3.name.to_owned(), s3);
```

Es werden, wie aus Abbildung 5.1 hervorgehend, drei Zustände benötigt. Daher werden in Zeile 1 bis 3 die drei Zustände über deren *new*-Methode erzeugt und an die Variablen *s1*, *s2* und *s3* gebunden. Die Zustände werden dann in die HashMap *states* eingefügt. Dabei ist der Name des jeweiligen Zustands der Schlüssel und der Zustand selbst, der Wert in der Map.

Nun können für die Zustände Transitionen festgelegt werden.

Listing 5.11: Definition der Transitionen und Erzeugung des Automaten

```
1         let mut transitions = HashMap::new();
2         transitions.insert(("s1".to_owned(), "go"), "s2".to_owned()
              );
3         transitions.insert(("s2".to_owned(), "back"), "s1".to_owned
              ());
4         transitions.insert(("s2".to_owned(), "finish"), "s3".
              to_owned());
5
6         let mut state_machine = StateMachine::new("s1".to_owned(),
              states, transitions);
```

Wie aus der Abb. 5.1 hervorgeht benötigen wir drei Transitionen. Zunächst wird eine HashMap *transitions* erstellt, in die anschließend die Transitionen eingefügt werden. Daher finden sich in Zeile 2 bis 4 die Definition der Transitionen. Dabei wird das Tupel aus Name des Status und Eingabezeichen als Schlüssel angegeben (zum Beispiel *('s1'.to_owned(), 'go')*, siehe Zeile 2).

In Zeile 6 wird der endliche Automat über die *new*-Methode aus den vorher erzeugten Komponenten erstellt. Dabei wird *S1* als Anfangsstatus angegeben, und die Maps der Status und Transitionen übergeben. Die Variable muss *mutable* sein, da unter anderem die next-Methode, die eine Zustandsänderung herbeiführt, innerhalb des endlichen Automaten dessen Attribute verändert.

Listing 5.12: Durchführung der Transitionen und sicherstellen der Korrektheit

```
1         state_machine.start();
2         state_machine.next("go");
3         state_machine.next("finish");
4
5         assert_eq!(21, context.borrow_mut().data);
6         assert_eq!("TestName", context.borrow_mut().name);
7         assert_eq!(true, state_machine.accepted);
```

Der konstruierte Automat kann nun verwendet und auf seine Korrektheit getestet werden. Zunächst wird dazu der endliche Automat durch Aufruf der *start*-Methode zur Ausführung gebracht und somit die Aktion des Startzustands ausgeführt, wobei in diesem Beispiel keine vorhanden ist. Anschließend wird über die next-Methode eine Transition ausgeführt. Dabei wird das Eingabezeichen 'go' übergeben. Danach sollte sich der Status in Zustand S2 befinden. Dazu sollte die Aktion des Zustands S2 ausgeführt werden, die die Attribute des Kontexts verändert.

Abschließend wird ein weiterer Zustandsübergang ausgeführt und 'finish' als Eingabezeichen übergeben. Der Automat sollte sich nun im akzeptierenden Zustand S3 befinden.

Zur Überprüfung der korrekten Ausführung des Automaten werden in Zeile 5 bis 6 die Attribute des Kontextes mit den erwarteten Werten verglichen. Abschließend wird validiert, ob sich der Automat in einem akzeptierenden Zustand befindet.
Die Verwendung des Automaten ist noch mit viel Aufwand verbunden. Im nächsten Schritt soll die Verwendung vereinfacht und gleichzeitig redundanter Code verhindert werden.

5.3 Einführung einer Domain Specific Language

Um den Benutzer die Verwendung des endlichen Automaten zu vereinfachen, wird im folgenden eine Domain Specific Language (kurz DSL) entwickelt und in Rust durch Makros implementiert.

Listing 5.13: Die DSL

```
1  fsm!(
2      States: {
3          (State1, false, Action1),
4          ...,
5          (State2, false, Action2)
6      }
7
8      Initial State: InitialState
9
10     Transitions: {
11         State1 => Token1 => State2,
12         ...,
13         State2 => Token2 => State1
14     }
15 )
```

Die DSL besteht aus drei Bestandteilen: Eine Menge an Zuständen (*States*), der Angabe des Startzustands (*Initial State*) und die Transitionen (*Transitions*). Ein Zustand aus der Menge der Zustände wird jeweils durch ein Tripel *(Name, akzeptierender Zustand?, Aktion)*, bestehend aus Name des Status, einem Flag, ob der Status akzeptierend ist, und der Aktion die bei Eintritt des Status ausgeführt werden soll, beschrieben. Transitionen werden durch die Schreibweise (*StatusX => Eingabezeichen => NachfolgenderZustand*) angegeben.

5.4 Implementierung in Rust durch Makros

Die DSL wird mit Hilfe von einem Makro umgesetzt. Dabei wird das Makro zur besseren Erklärung in seine Bestandteile aufgeteilt. Zunächst folgt der Pattern Matching-Teil des Makros.

Listing 5.14: Der Kopf des Makros

```
1  macro_rules! fsm (
2      (
3          States: {
4              $(($name:ident, $is_accepting: expr, $action: expr)), +
5          }
6
7          Initial State: $initial_state:ident
8
9          Transitions: {
10             $($current_state: ident => $token:expr =>
                   $following_state:ident), *
11         }
12     ) => {
```

Der Aufbau der DSL kann relativ einfach in ein Pattern für ein Rust Makro umgesetzt werden. In Zeile 3 bis 5 findet die Definition der Zustände statt. Zeile 4 ist dabei am wichtigsten. Hier findet sich eine Wiederholung wieder, die mindestens ein Element haben muss (ausgedrückt durch das +). In der Wiederholung wird der Name eines Status an die Variable *$name* gebunden. Darüber hinaus wird das Flag, ob der Status akzeptierend ist, an die Variable *$is_accepting*, und die Aktion des Status an die Variable *$action* gebunden.

In Zeile 7 wird der Startzustand festgelegt, der an die Variable *$initial_state* gebunden wird.

In Zeile 9 bis 11 werden die Transitionen erfasst. Zunächst haben wir eine Wiederholung. In dieser wird an erster Stelle ein Status angegeben und darauf folgt die Zeichenfolge =>. Dann wird ein Token angegeben, das an die Variable *$token* gebunden wird und nach einer weiteren Zeichenfolge => folgt der nachfolgende Status, der an die Variable *$following_state* gebunden wird.

Matcht das Pattern, so wird der folgende Quellcode durch das Makro eingefügt.

Listing 5.15: Die Konstruktion der Zustände

```
1          {
2              let mut i = 1;
3              $(
```

```
4                println!("State: {}, finite?: {}, Action: {}",
                     stringify!($name), stringify!($is_accepting),
                     stringify!($action));
5                i += 1;
6            )+
7
8            let mut states: HashMap<String, State> = HashMap::
                 with_capacity(i);
9            $(
10               println!("State: {}, finite?: {}, Action: {}",
                     stringify!($name), stringify!($is_accepting),
                     stringify!($action));
11               let $name = State::new($is_accepting, stringify!(
                     $name), $action);
12               states.insert($name.name.to_owned(), $name);
13           )+
```

Zunächst wird in Zeile 2 eine Hilfsvariable i eingeführt, die zum Zählen der Zustände genutzt wird. Dies wird in den Zeilen 3 bis 6 durchgeführt, indem in der Wiederholung die Variable i inkrementiert wird. Die nun bekannte Anzahl der Status wird in Zeile 8 genutzt, um eine HashMap *states* mit der Kapazität i zu erzeugen (Methode with_capacity(i)). Die HashMap mappt Strings auf States, also den Namen eines Zustands auf das eigentliche Zustands-Objekt.

In Zeile 9 bis 13 findet sich erneut eine Wiederholung, die zunächst über *println!* die Eigenschaften eines Zustands, also dessen Name, Flag akzeptierender Zustand und Aktion, auf der Standardausgabe ausgibt.[2] Danach wird in Zeile 11 eine State-Instanz über die statische Methode *new* der Struktur State erzeugt, die als Parameter den Name (*$name*), das Flag, akzeptierender Zustand (*$is_accepting*), und das Aktions-Objekt (*$action*) erhält. Als letzte Anweisung in der Wiederholung wird die zuvor erzeugte Status-Instanz in die zuvor erstellte HashMap *states* eingefügt.

Die Zustände werden nun zur Konstruktion der Transitionen genutzt.

Listing 5.16: Die Konstruktion der Transtitionen

```
1            let mut transitions = HashMap::new();
2            $(
3                println!("{} => {} => {}", stringify!(
                     $current_state), stringify!($token), stringify
                     !($following_state));
```

[2] Der Inhalt von Variablen in Makros kann über das Makro *stringify!* in einen String umgewandelt werden

```
4                   transitions.insert((stringify!($current_state).
                        to_owned(), $token), stringify!(
                        $following_state).to_owned());
5               )+
6
7               let sm = StateMachine::new(stringify!($initial_state).
                    to_owned(), states, transitions);
8               sm
9           }
10      };
11  );
```

Zunächst wird in Zeile 1 eine veränderbare HashMap *transitions* erzeugt. Diese wird dann in Zeile 2 bis 5 mit den Transitionen gefüllt. Als erstes werden in Zeile 3 die Transitionen auf der Standardausgabe ausgegeben. In der Zeile 4 wird über die *insert*-Methode der HashMap ein Tupel bestehend aus *(String, Typ des Tokens)* als Schlüssel und einem *String* als Wert in die Map eingefügt. Das Tupel *(String, Typ des Tokens)* repräsentiert die dabei *(Name des Status, Eingabezeichen)* und der Wertes der HashMap den Namen des nachfolgenden Status.

In Zeile 7 findet die eigentliche Konstruktion des endlichen Automaten statt. Dabei wird die *new*-Methode der *StateMachine*-Struktur genutzt, um den Automaten zu erzeugen. Als Parameter wird der initiale Zustand, der an die *$initial_state* Variable gebunden ist, übergeben. Darüber hinaus werden noch die zuvor konstruierten Maps *states* und *transitions* weitergegeben.

Das Makros ist stark an die Implementation der *StateMachine*-Struktur gebunden. Ändert sich die *new*-Methode, so funktioniert auch das Makro nicht mehr. Daher muss man immer abwägen, ob sich eine Implementation eines Makros lohnt.

5.5 Test des Makros

Im Folgenden wird das zuvor erstelle Makro auf seine Richtigkeit getestet. Gleichzeitig wird eine beispielhafte Verwendung des Makros dargestellt. Dabei wird der Automat aus Abb. 5.1 wieder als Grundlage genommen.

In diesem Beispiel wird ein Enum statt Strings zur Definition der Eingabezeichen benutzt.

Listing 5.17: Die Eingabezeichen

```
1   #[derive(PartialEq, Eq, Hash)]
2   enum Token {
3       Go,
```

```
4         Back,
5         Finish
6     }
```

Der Enum *Token* umfasst drei Varianten (*Go*, *Back* und *Finish*). Die Varianten repräsentieren die Eingabezeichen des endlichen Automaten aus Abb. 5.1. Dieser Enum kann dann nachfolgend zur Definition der Transitionen genutzt werden. Über das derive-Attribut werden die Traits PartialEq, Eq und Hash bei der Kompilierung durch den Compiler implementiert. Diese werden benötigt, damit die Varianten des Enum gehasht werden können.

Die Definition des Kontexts und der Aktionen sind äquivalent zu dem Test des ersten Ansatzes und werden daher ausgelassen. Darauf folgt die Verwendung des Makros.

Listing 5.18: Verwendung des Makros

```
1         let mut state_machine = fsm!(
2             States: {
3                 (s1, false, None),
4                 (s2, false, Some(s2_action)),
5                 (s3, true, Some(s3_action))
6             }
7
8             Initial State: s1
9
10            Transitions: {
11                s1 => Token::Go => s2,
12                s2 => Token::Back => s1,
13                s2 => Token::Finish => s3
14            }
15        );
```

Zunächst wird in Zeile 1 die Variable *state_machine* definiert, an die der durch das Makro erzeugte Automat gebunden wird. Die Variable muss wie auch im vorherigen Test mutable sein, um Zustandsänderungen durchzuführen. Der Aufruf des Makros befindet sich in Zeile 1 bis 15. Dabei ist die Struktur aus der vorher definierten DSL wiederzufinden. Zunächst werden in Zeile 2 bis 6 die drei Zustände (*s1*, *s2* und *s3*) definiert und dabei wieder angegeben, ob diese akzeptierend sind und, wenn vorhanden, eine Aktion übergeben. In Zeile 8 wird der Startzustand angegeben (hier *s1*). In den Zeilen 10 bis 14 werden die drei Transitionen definiert, welche wieder auf den endlichen Automat aus Abb. 5.1 hervorgehen. Als Eingabezeichen werden die Varianten des *Token* Enums verwendet.

Danach ist die Konstruktion des endlichen Automaten durch das Makro abgeschlossen und dieser kann nun verwendet werden.

Listing 5.19: Benutzung des endlichen Automaten

```
1         state_machine.start();
2         state_machine.next(Token::Go);
3         state_machine.next(Token::Finish);
4
5         assert_eq!(21, context.borrow_mut().data);
6         assert_eq!("TestName", context.borrow_mut().name);
7         assert_eq!(true, state_machine.accepted);
```

Dazu wird der endliche Automat durch die start-Methode zunächst in den Startzustand versetzt. Danach werden wieder zwei Transitionen durchgeführt. Zuallererst wird der Automat durch das Eingabezeichen *Token::Go* von S1 in S2 und anschließend durch das Eingabezeichen *Token::Finish* von S2 in S3 überführt. Darauf werden zur Überprüfung der Korrektur des Makros die Attribute des Kontextes mit den zu erwartenden werten verglichen. Darüber hinaus wird sichergestellt, ob sich der Automat am Ende in einem akzeptierenden Zustand befindet.

Die Konstruktion des Automaten wurde durch die Einführung der DSL und der Verwendung des Makros vereinfacht.

5.6 Bewertung der Implementation

Im Folgenden wird die Implementation, die im Laufe dieser Arbeit erstellt wurde, bewertet und gleichzeitig Vergleiche mit anderen Implementationen von Automaten verglichen.

5.6.1 Hierarchie

Der in dieser Arbeit implementierte Automat stellt eine sehr grundlegende Art eines Moore-Automaten dar. Harel hat in seinem Paper (siehe [9]) *Statecharts* eingeführt, die komplexe Systeme durch visuelle Formalisierungen realisierbar machen sollen. Diese Statecharts sind erweiterte Automaten, die Eigenschaften implementieren, die in reaktiven Systemen von Nutzen sind. Das Konzept geht vor allem darauf zurück, dass man hierarchische Automaten erstellen kann, die über verschiedene Ebenen verfügen und somit die Modellierung von großen Systemen übersichtlicher gestaltet. In dieser Arbeit wurde ein flacher, endlicher Automat implementiert, der über keinerlei Hierarchie verfügt. Da die Logik der Transitionen in dem Automaten und nicht in seinen Zuständen gelagert ist, müsste man zum hinzufügen einer Hierarchie den Automaten, nicht aber seine Zustände ändern. Man könnte auch eine weitere

Schicht über den flachen Automaten setzen, die eine Hierarchie emuliert. Dies würde aber zum einen zu einem exponentiellen Anstieg des Platzverbrauch führen und darüber hinaus wäre die Wartbarkeit eines solchen Automaten erschwert, da der eigentliche Automat unter der Abstraktionsschicht sehr groß und unübersichtlich werden würde (vgl. [9]). Zusätzlich zur Hierarchie besitzen Statecharts eine so genannte *History*[9, S.238]. Durch diese wird ermöglicht, dass bei Wiedereintritt in einen übergeordneten Zustand der zuletzt betretene innere Status gewählt wird. Alternativ dazu kann dies auch eine standardmäßiger (*default*) Status sein. Der Automat aus der Arbeit verfügt nicht über eine History-Funktion, da er auch keinerlei Hierarchie besitzt. Diese Funktion sollte daher mit der Hierarchie implementiert werden.

5.6.2 Aktionen

Die Aktionen eines Automaten müssen beendet werden, damit der Automat in den nächsten Zustand wechseln kann. Daher ist es nicht empfehlenswert, größere Berechnungen in den Aktionen durchzuführen, wenn diese im gleichen Thread stattfinden.

Darüber hinaus ist der Implementation eines endlichen Automaten nur eine Eingangsaktion vorhanden, die ausgeführt wird, sobald ein Status betreten wird. Zusätzlich kann eine Ausgangsaktion hinzugefügt werden. Dazu muss die Status-Struktur um eine zusätzliche Aktion erweitert werden. Des Weiteren muss das Ausführen der Austrittsaktion zu der *next*-Methode des endlichen Automaten hinzugefügt werden.

Da der Automat nicht an Mealy (vgl. [17, S.28]) gelehnt ist, besitzt der Automat keine Aktionen, die an Transitionen gebunden sind. Dazu könnte die Übergangstabelle erweitert werden, sodass sie zusätzlich zur Transition eine Aktion speichert.

5.6.3 Kontext

Im Test wurde dargestellt, wie innerhalb des Automaten zusätzliche Informationen über einen Kontext gespeichert werden können. Dieser Kontext kann auch direkt in der Implementation des Automaten untergebracht werden. GoF (siehe [7], S.400) zeigt in seinem Zustandsmuster, wie der Kontext in dem Automaten untergebracht werden kann. Eine Schwierigkeit stellt dabei die zyklische Abhängigkeit von Kontext und Status dar. Dies kann gerade bei Rust Probleme bieten, da durch gegenseitiges Borrowing von Variablen eine Art Deadlock entsteht. Die Rust FAQ ([22], siehe *Ownership*) bietet Ansätze zum Lösen dieses Problems.

5.6.4 Parallelität

Harel[9, S.242ff] spricht darüber hinaus die Parallelität und Unabhängigkeit (*Orthogonality*) als Eigenschaften von Statecharts an. Dies sagt aus, dass sich der Automat in mehreren Zuständen befinden kann, deren Aktionen parallel ausgeführt werden können. Somit besteht der eigentliche Zustand eines Automaten immer aus einer Kombination von Zuständen. Die Unabhängigkeit ist auf die Zustände einer jeweiligen Kombination bezogen. Damit ist gemein, dass wenn ein Event eintritt, so müssen nicht alle Zustände eine Transition ausführen. Die Implementation des Automaten aus der Arbeit verfügt über keinerlei Parallelität in den Zuständen. Diese müsste zusätzlich zur Hierarchie eingebaut werden.

5.6.5 Akzeptierender Zustand

Die Handhabung eines erreichten akzeptierten Zustands stellt einen wichtigen Punkt in der Implementation eines Automaten dar. In der Implementation aus der Arbeit gibt es dafür das *accepted*-Flag, dass zur Abfrage genutzt werden kann. Somit muss mittels Polling (zum Beispiel in einer while-Schleife) abgefragt werden, ob ein akzeptierender Zustand eingetreten ist.

Ein anderer Ansatz wäre die Benachrichtigung durch eine Nachricht oder ähnliches bei Eintritt in einen akzeptierenden Zustand.

5.6.6 Fehlerbehandlung

Die Fehlerbehandlung in der vorliegenden Implementation ist auf das Nötigste reduziert, um die eigentliche Logik der Implementation möglichst simpel zu halten, und damit den Quellcode lesbar zu halten. Daher wird die Methode *unwrap* des Option-Enum genutzt. Somit führt eine fehlerhafte Benutzung des Automaten direkt zum Absturz, wobei eine generische Fehlermeldung ausgegeben wird. Besser wäre es, die unwrap-Methode durch eine eigene Fehlerbehandlung zu ersetzen, die zumindest eine passende Fehlermeldung ausgibt.

Ein Weiterer Punkt ist die Handhabung von nicht vorhandenen Transitionen. Hier bestehen auch mehrere Möglichkeiten dieser. Zum einen kann dazu in einen Fehlerzustand gegangen werden. Durch Abfrage dieses Status können weitere Operationen durchgeführt werden. Eine weitere Möglichkeit ist das Stoppen des Programms bei einer nicht vorhandenen Transition, welches aber einen sehr harten Eingriff in den Ablauf des Programms stellt und sollte daher gut abgewogen sein. Die Implementation aus der Arbeit behält bei nicht vorhandener Transition den gleichen Zustand bei.

5.6.7 Platzverbrauch

Der Platzverbrauch stellt gerade in eingebetteten Systemen zum Teil ein kritisches Thema dar, wenn nicht genug Ressourcen vorhanden sind. Oft verkörpert eine Lösung einen Trade-off zwischen Lesbarkeit und Optimierung.

In der erarbeiteten Implementation werden alle Zustände bei der Definition des endlichen Automaten erzeugt. Diese werden bis zum Ende der Lebenszeit des Automaten im Speicher gehalten. Dies hat den Vorteil, dass nach der erstmaligen Erzeugung des Automaten keine Weitere Zeit für die Erzeugung von Zuständen beansprucht wird. Eine weitere Möglichkeit wäre es, die Erzeugung eines Status erst durchzuführen, wenn dieser benötigt wird. Dadurch könnte es allerdings zu Verzögerungen kommen, da möglicherweise vor einer Transition zunächst Speicherplatz alloziert und ein Zustand erzeugt werden müsste. In der Implementationen werden die Zustände durch Instanzen einer Struktur dargestellt. Ein anderer Ansatz wäre die Verwendung von Enums. Dadurch würde sich der Speicherverbrauch enorm reduzieren, da nur noch das Tag benötigt wird, dass die Variante des Enums angibt. Dabei richtet sich die Größe des Speicherplatz, die ein Enum verbraucht, nach der größten Variante des Enums [14]. Eine beispielhafte, aber veraltete Version findet sich bei GitHub[3]. Daraus folgt, dass der verbrauchte Speicher in Hinblick auf die Zustände konstant ist.

Darüber hinaus wird in der Implementation fast ausschließlich der Datentyp *String*[29] aus der Standardbibliothek benutzt. Besser ist stattdessen, sofern möglich, den Datentyp *&str* zu benutzen, der einen Ausschnitt auf einen String darstellt[22]. Dadurch wird der belegte Speicher um einen großen Teil verringert, da nur noch ein String benötigt wird und dieser durch &str referenziert werden kann. Auf Verwendung von &str wurde während der Implementation verzichtet, da dies die Implementation verkomplizieren und schlechter lesbarer machen würde. Durch Verwendung von Referenzen müssten die Deklarationen zusätzlich durch *Lifetimes* annotiert werden, damit der Compiler verifizieren kann, dass die Referenzen lange genug leben.

[3] Siehe https://github.com/thehydroimpulse/rust-fsm/blob/master/src/lib.rs

6 Zusammenfassung

In dieser Bachelorarbeit wurden unter anderem moderne Entwurfsmuster auf ihre Übertragbarkeit in Rust untersucht, um die Plattform zur Sammlung von Mustern und anderen Elementen, die in der systemnahen Programmierung Gebrauch finden, zu erweitern.

Dabei wurde zunächst die Plattform vorgestellt und anschließend in die Programmiersprache Rust eingeführt. Dabei wurden Grundlagen für die Implementationen, die im Verlauf der Bachelorarbeit entstanden sind, gelegt.

Zunächst wurde das Builder Pattern nach GoF umgesetzt. Das Muster konnte relativ einfach in Rust übersetzt werden. Dabei konnten die Elemente, wie der Director, Builder usw., gut auf die sprachlichen Elemente von Rust übertragen werden. Darüber hinaus wurde das Builder Pattern begutachtet, welches in der Rust Dokumentation empfohlen wird. Durch dieses kann mittels Method-Chaining die fehlende Möglichkeit zur Überladung von Funktionen/Methoden ausgeglichen werden. Allerdings eignet sich dieser Ansatz auch aufgrund des Method-Chainings nicht zur Erzeugung komplexer Objekte. Hier ist die Umsetzung nach GoF geeigneter.

Des Weiteren wurde eine Abstract Factory in Rust implementiert. Die Elemente dieses Musters konnten ebenfalls gut auf die sprachlichen Elemente von Rust übertragen werden.

Darüber hinaus wurde das Factory Pattern in Rust umgesetzt. Dabei wurde mit Hilfe eines Makros ein statischer Ansatz implementiert, der kein Hinzufügen von Produkten zur Laufzeit anbietet. In diesem Ansatz wurde die praktische Verwendung von Makros zur Vermeidung von Redundanzen aufgezeigt.

Alles in allem kann man sagen, dass die modernen Entwurfsmuster in Rust übersetzbar sind. Allerdings bleibt abzuwägen, ob eine direkte Umformulierung sinnvoll ist oder es andere und besser geeignetere Ansätze gibt.

Darüber hinaus wurde in dieser Bachelorarbeit ein deterministischer endlicher Automat implementiert, der an die Moore-Implementation angelehnt ist. Dabei wurden zur Speicherung der Zustände und Transitionen Hash Maps genutzt. Zur besseren Verwendung und Vermeidung von redundantem Code, wurde ein Makro nach einer zuvor erstellten DSL umgesetzt.

Abschließend wurde der Automat bewertet und weitere Möglichkeiten zur Veränderung des Automaten aufgezeigt.

7 Ausblick

Rust ist im Vergleich zu Sprachen wie C/C++ noch sehr jung. C/C++ werden schon seit Jahrzehnten produktiv eingesetzt und besitzen in der systemnahen und eingebetteten Programmierung einen riesigen Marktanteil. Im Gegensatz dazu ist die erste stabile Version von Rust im Frühjahr 2015 erschienen[20].

Rust bietet viele interessante Konzepte und es bleibt abzuwarten, ob sich diese auch über einen längeren Zeitraum in realen Softwareprojekten als produktiv erweisen. Die Sicherheit in Softwaresystemen wird immer wichtiger. Durch Konzepte wie dem Ownership-Modell und dem Borrowing verlagert Rust die Probleme wie Buffer Overflows und Segmentation Faults vom Programmierer zum Compiler, der diese sicher analysieren kann.

Darüber hinaus bestehen bereits Ausarbeitungen, um das Konzept der Lifetimes auch in C++ umzusetzen[19], was darauf hindeutet, dass die Konzepte einen sinnvollen Ansatz zur sicheren Speicherverwaltung darstellen.

Das Team von Rust hat letztes Jahr bereits anstehende Punkte für die Weiterentwicklung von Rust im Jahr 2016 veröffentlicht[15]. Einer der Punkte ist, die Versionsübergänge von Rust möglichst problemfrei durchzuführen. Daher werden neue Versionen automatisiert durch ein Tool gegen die Bibliotheken, die sich auf der crates.io Seite befinden, getestet. Darüber hinaus stellt die Verbesserung der Kompilezeiten einen wichtigen Punkt dar. Durch inkrementelle Kompilation sollen diese verbessert werden. Ein weiterer wichtiger Punkt, um die Sprache zu etablieren, stellt zum einen die Verbesserung der IDE Integration und zum anderen die Vereinfachung des Cross-Kompilierens dar. Dadurch können mehr Plattformen einfach erreicht werden und somit würde sich auch die Programmierung für eingebettete Systeme erleichtern. Um einen größeren Markt zu erreichen, sollte Rust möglich einfach und komfortabel nutzbar sein.

Gerade in Hinblick auf das Internet of Things hat Rust gute Chancen auf dem Markt zu konkurrieren. Immer mehr Geräte werden intelligent und sollen möglichst stabil und unabhängig laufen, nachdem sie einmal installiert wurden.

Literaturverzeichnis

[1] : *C++ - shared_ptr*. – URL http://de.cppreference.com/w/cpp/memory/shared_ptr. – Zugriffsdatum: 2016-02-17

[2] : *Can't define a function name using concat_idents!()*. – URL https://github.com/rust-lang/rust/issues/12249. – Zugriffsdatum: 2016-04-02

[3] : *CWE-416: Use After Free*. – URL https://cwe.mitre.org/data/definitions/416.html. – Zugriffsdatum: 2016-04-03

[4] Chen, Franklin: *How to think about Rust ownership versus C unique_ptr*. 2016. – URL http://conscientiousprogrammer.com/blog/2014/12/21/how-to-think-about-rust-ownership-versus-c-plus-plus\-unique-ptr/. – Zugriffsdatum: 2016-02-17

[5] Eilbrecht ; Starke: *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. Springer Vieweg, 2013

[6] Free Software Foundation: *C++ - Macros*. 2016. – URL https://gcc.gnu.org/onlinedocs/cpp/Macros.html. – Zugriffsdatum: 2016-02-22

[7] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. 5. Addison-Wesley, 2001

[8] Gartner, Inc.: *Gartner Says 6.4 Billion Connected "Things"Will Be in Use in 2016, Up 30 Percent From 2015*. 2015. – URL http://www.gartner.com/newsroom/id/3165317. – Zugriffsdatum: 2016-03-22

[9] Harel, David: Statecharts: A Visual Formalism For Complex Systems. (1984)

[10] Hopcroft, John E. ; Motwani, Rajeev ; Ullman, Jeffrey D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. 2. Addison-Wesley, 2002

[11] Katz, Yehuda: *The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire.* 2014. – URL http://blog.skylight.io/bending-the-curve-writing-safe-fast-native-gems-with\-rust/. – Zugriffsdatum: 2016-02-16

[12] Kohlbecker, Eugene E.: Macro-By-Example: Deriving Syntactic Transormation from their Specifications.

[13] Maidsafe: *Why switch from C++ to Rust?* 2015. – URL https://safenetwork.wiki/en/FAQ#Why_switch_from_C.2B.2B_to_Rust.3F. – Zugriffsdatum: 2016-02-16

[14] Matsakis, Nicholas D.: *Virtual Structs Part 1: Where Rust's Enum Shines.* 2016. – URL http://smallcultfollowing.com/babysteps/blog/2015/05/05/where-rusts-enum-shines/. – Zugriffsdatum: 2016-02-17

[15] Matsakis, Nicholas D. ; Turon, Aaron: *Rust in 2016.* 2015. – URL http://blog.rust-lang.org/2015/08/14/Next-year.html. – Zugriffsdatum: 2016-02-17

[16] Metz, Cade: *The Epic Story of DropboxâĂŹs Exodus From the Amazon Cloud Empire.* 2016. – URL http://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/. – Zugriffsdatum: 2016-02-16

[17] Samek, Miro: *Practical statecharts in C/C++ : quantum programming for embedded systems.* CMP Books, 2002

[18] Stroustrup, Bjarne: *The Design and Evolution of C++.* Addison-Wesley, 1994

[19] Sutter, Herb ; MacIntosh, Neil: *Lifetime Safety: Preventing Leaks and Dangling.* 2015. – URL https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetimes%20I%20and%20II%20-%20v0.9.1.pdf. – Zugriffsdatum: 2016-02-22

[20] The Rust Core Team: *Announcing Rust 1.0.* 2015. – URL http://blog.rust-lang.org/2015/05/15/Rust-1.0.html. – Zugriffsdatum: 2016-02-17

[21] The Rust Project Developers: *Cargo Guide.* 2016. – URL http://doc.crates.io/guide.html. – Zugriffsdatum: 2016-02-16

[22] The Rust Project Developers: *Rust - Frequently Asked Questions*. 2016. – URL https://www.rust-lang.org/faq.html. – Zugriffsdatum: 2016-02-17

[23] The Rust Project Developers: *The Rust Programming Language*. 2016. – URL https://doc.rust-lang.org/stable/book/. – Zugriffsdatum: 2016-02-16

[24] The Rust Project Developers: *The Rust Reference*. 2016. – URL https://doc.rust-lang.org/reference.htm. – Zugriffsdatum: 2016-04-05

[25] The Rust Project Developers: *Struct std::boxed::Box*. 2016. – URL https://doc.rust-lang.org/std/boxed/struct.Box.html. – Zugriffsdatum: 2016-02-17

[26] The Rust Project Developers: *Struct std::cell::RefCell*. 2016. – URL https://doc.rust-lang.org/std/cell/struct.RefCell.html. – Zugriffsdatum: 2016-02-17

[27] The Rust Project Developers: *Struct std::option::Option*. 2016. – URL https://doc.rust-lang.org/std/option/enum.Option.html. – Zugriffsdatum: 2016-02-17

[28] The Rust Project Developers: *Struct std::rc::Rc*. 2016. – URL https://doc.rust-lang.org/std/rc/struct.Rc.html. – Zugriffsdatum: 2016-02-17

[29] The Rust Project Developers: *Struct std::string::String*. 2016. – URL https://doc.rust-lang.org/std/string/struct.String.html. – Zugriffsdatum: 2016-02-17

[30] Turon, Aaron: *Abstraction without overhead: traits in Rust*. 2015. – URL http://blog.rust-lang.org/2015/05/11/traits.html. – Zugriffsdatum: 2016-02-17

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 14. April 2016 Felix Runge