Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Verwendung der JAVA API Graphics2D zur Bildkomposition durch Implementierung einer domänenspezifischen Sprache in Clojure

Nicolas Schwartau

2117446

Hochschule für Angewandte Wissenschaften Hamburg

Fakultät Design, Medien und Information

Department Medientechnik

Studiengang: Media Systems

Erstprüfer: Prof. Dr. Edmund Weitz

Zweitprüfer: Prof. Dr. Andreas Plaß

Abgabetermin: 30.05.2016

Sommersemester 2016

Hamburg, 28.05.2016

Abstract

Effective Bodyweight Training GmbH is a startup company offering an app for functional training with body weight only. The app provides features like doing workouts, saving achieved results and comparing them to other users results on a leader board. The app is available for the mobile platforms Android and iOS. The backend is written in Clojure. The goal of Effective Bodyweight Training GmbH is to generate a higher user range with the establishment of a new feature in order to differentiate from other competing companies in this market. For this reason, users should be able to share workout results on Facebook with their friends. Relating to the comparison of workout results with friends via Facebook messages an image needs to be created. The image must display information about the latest workout details as wells as results and the user having achieved this results.

After a preceding search, it turned out that there are currently no tools in Clojure that meet the company's requirements. Being an employee of Effective Bodyweight Training GmbH the instruction was to create a program, being able to compose such an image. It is likely, that this task provides vast potential for an open source project potentially being interesting for other developers as well. Therefore, the decision was made to develop a domain specific language, in short DSL, in Clojure. While developing the DSL the questions will be answered whether the programming language Clojure is suitable for developing a DSL and whether a functional programming language contributes in an easier manner to the creation of applications implementing multithreading features.

Inhaltsverzeichnis

Inhaltsverzeichnis III
Glossar V
Abkürzungsverzeichnis VII
Tabellenverzeichnis VIII
Abbildungsverzeichnis IX
Listingverzeichnis X
1. Motivation 1
2. Zielsetzung 2
3. Grundlagen von Clojure 3
3.1. Funktionale Programmierung 3
3.1.1. State 4
3.1.2. Threads und das Problem State 5
3.1.3. Lösungsweg in Clojure 5
3.2. Syntax 6
3.2.1. Special Forms 7
3.2.2. Funktionen 8
3.3. Datenstrukturen 9
3.3.1. Symbols und Keywords 9
3.3.2. List 10
3.3.3. Vector 11
3.3.4. Map 11
3.3.5. Sequence 12
3.3.6. Persistent Datastructure 12
3.3.7. Managed Reference 13
3.4. Funktionen höherer Ordnung 13
3.5. Java Interoperabilität 14
3.6. Makro-System 15
3.7. Closures 17
4. Grundlagen der DSL 18
4.1. Domain Specific Language 18
4.2. Java 2D API 21
4.2.1. User Space 21
4.2.2. Aufgaben 23

4.2.3. Die Klassen Graphics und Graphics2D ... 24
5. DSL Image-Compojure ... 28
5.1. Anforderungen ... 28
5.1.1. Gegenstand (Was?) ... 28
5.1.2. Verhalten (Wie?) ... 29
5.2. Grammatik und Verwendung ... 31
5.2.1. Compose ... 31
5.2.2. Shapes ... 33
5.2.3. Farben, Verläufe, Texturen ... 34
5.2.4. With-attributes ... 35
5.2.5. With-transform ... 37
5.2.6. Bilder ... 38
5.2.7. Schriftzüge ... 39
5.2.8. Ausgabe ... 40
5.2.9. Uneingeschränkte Clojure Funktionalität ... 41
5.2.10. Veröffentlichung ... 42
5.3. Design und Implementierung ... 43
5.3.1. Konstrukt ... 43
5.3.2. Maps, Keys und Values ... 45
5.3.3. Funktionen für Shapes ... 46
5.3.4. Schriftzüge ... 48
5.4. Ausblick ... 49
5.4.1. Weiterentwicklung ... 49
5.4.2. Verwendung bei AnyUp ... 50
6. Fazit und Zusammenfassung ... 52
A. Material ... 54
A.1 Quellcode Composite Beispiel ... 54
A.2 Erstellen von Paint Values ... 55
A.3 Beispiel von simplen Zeicheinoperationen ... 55
A.4. Attribute für Schriftzüge ... 56
A.5. Abbildung zur Verwendung create-styled-text ... 57
A.6. Quellcode zur Abbildung AnyUp Open Graph ... 58
Literaturverzeichnis ... 59
Eigenständigkeitserklärung ... 61

Glossar

Durch einen (*) gekennzeichnete Begriffe wurden im Zuge der Entwicklung eigens erdacht.

AnyUp		Geschäftsbezeichnung der Firma Effective Bodyweight Training
Collection	Sammlung	Datenstrukturen, die eine Sammlung von Werten bilden
Concurrency Semantics	Semantik für Nebenläufigkeit	Logik für Programme zur Verarbeitung von parallelen Threads
Default-Vars*	Standard-Vars	Variablen der Standard-Attribut-Werte für Shape-Objekte und Render-Settings
Dependencies	Abhängigkeiten	Abhängigkeiten eines Projekts von anderen Projekten/ Programmen
Destructuring	Zerstörung	Zuweisung von Werten aus der Datenstruktur Map zu gleichnamigen Symbols
Dirty reads	Dreckige Lesezugriff	Lesen von inkonsistenten Daten
Device Space	Geräte Raum	Koordinatensystem des Ausgabemediums zur Darstellung eines Rendering-Models
Exception	Ausnahme	Darstellung von Fehlerzuständen im Programm
Floating Point	Fließkomma Zahlen	Datentypen wie float oder double
Free Variables	Freie Variablen	Stehen Funktionen weder als Argument noch als lokale Variable zur Verfügung
Homoikonizität		Quellcode einer Programmiersprache, stellt eine Datenstruktur der Sprache selbst dar
Image Space	Bild Raum	Lokales Koordinatensystem eines Bildes, in dem die Bounding Box zur Bildauswahl definiert wird
Immutable	Veränderbar	Datentypen, deren Werte im Speicher selbst nicht verändert/ gelöscht werden können
Key	Schlüssel	Zugriffsfunktion auf Werte des Datentyps Map
Multithreading		Verwendung von multiplen Threads zur Bewältigung einer Aufgabe
Mutable	Veränderbar	Datentypen, deren Werte im Speicher selbst verändert/ gelöscht werden können
Namespace		Scope, in dem eine Zuweisung zwischen einem Symbol und einem Var gültig ist
Phantom reads	Phantom Lesezugriffe	Lesen von Daten, die bereits gelöscht wurden.
Polygon		Form, die aus einer beliebigen Anzahl von Punkten besteht
Pure Function	Reine Funktion	Funktionen, die keine Nebeneffekte besitzen.
Referential Transparency	Referentielle Transparenz	Veränderte Daten erhalten neben dem neuen, auch den alten Stand

Rendering-Engine		Bestimmt den Algorithmus während des Renderings
Rendering-Operation		Funktionen, die Formen, Schriftzüge oder Bilder, auf eine Zeichenfläche zeichnen
Render-Settings*	Render-Optionen	Werte, die zur Definition der Rendering-Engine verwendet werden
Scope	Kontext	Gültigkeitsbereich eines Namespaces
Shape-Attribute*	Form-Attribut	Attribut-Werte, die das Aussehen eines Shape-Objekts auf einer Zeichenfläche bestimmen
Shape	Form	Objekt der Klasse *java.awt.geom*, gibt eine geometrische Form an, die auf eine Zeichenfläche gezeichnet wird
Side Effects	Nebeneffekte	Auswirkungen, die eine Funktion neben der Generierung des eigentlichen Rückgabewert besitzt
State	Zustand	Derzeitige Eigenschaften eines Objekts
S-Expression	S-Ausdruck	Verschachtelte Ausdrücke / Listen
Thread		Bestandteil eines Prozesses zur Bewältigung einer Aufgabe
Unique	Einzigartig	Eindeutigkeit eines Wertes in einer Collection
User Space	Benutzer Raum	Koordinatensystem, in dem angegeben wird an welcher Stelle sich Formen o.Ä. befinden
Value	Wert	Wert zu einem Key in der Datenstruktur Map

Abkürzungsverzeichnis

API	Application Programming Interface
APP	Applikation
DSL	Domain Specific Language
G2D	Graphics2D
GPL	General Pupose Language
JVM	Java Virtual Machine
OO	objektorientiert
SF	Special Form
SQL	Structured Query Language

Tabellenverzeichnis

Tabelle 5.1 Render-Settings 32
Tabelle 5.2 Shape-Attribute 36
Tabelle A.1 Attribute für Schrift 56

Abbildungsverzeichnis

Abbildung 3.1 Symbol xs mit neuer Abstraktion ys (Quelle: Clojure in Action, 2016, S. 5)....4
Abbildung 3.2 Verschachtelte Listen (angelehnt an: Clojure in Action, 2016 S. 10)....6
Abbildung 3.3 Phasen zur Clojure Laufzeit (Quelle: Clojure in Action, 2016, S. 167) 15
Abbildung 4.1 Rechteck am Ursprung....22
Abbildung 4.2 Schrift-Baseline (angelehnt an Oracle JAVA docs, 05.05.2016, Measuring Text) 22
Abbildung 4.3 Beispiel Translation 25
Abbildung 4.4 Beispiele Composite....26
Abbildung 5.1 Beispiel Clojure Funktionalität 42
Abbildung 5.2 AnyUp Open Graph 50
Abbildung A.1 Erstellung von Paint Values 55
Abbildung A.2 Beispiel simpler Rendering-Operationen....55
Abbildung A.3 Verwendung create-styled-text 57

Listingverzeichnis

Listing 3.1 If-Bedingung 7
Listing 3.2 Funktionen definieren 8
Listing 3.3 Lists 10
Listing 3.4 Vektoren 11
Listing 3.5 Maps 11
Listing 3.6 Destructuring (angelehnt an Clojure in Action, 2016, S. 95) 12
Listing 3.7 Var Binding 13
Listing 3.8 Interop Beispiel 14
Listing 3.9 Makro Beispiel (angelehnt an Clojure in Action, 2016, S. 180) 16
Listing 3.10 Closures 17
Listing 4.1 Assoziationen in Rails 20
Listing 4.2 Programmcode zu Abbildung 4.1 23
Listing 4.3 Programmcode zu Abbildung 4.3 25
Listing 5.1 Anwendung compose Makro 32
Listing 5.2 Beispiel shape Funktion 33
Listing 5.3 Erstellung von Paint Values 34
Listing 5.4 Anwendung with-attributes Makro 35
Listing 5.5 Beispiel simpler Rendering-Operationen 36
Listing 5.6 Transformationsbespiel 37
Listing 5.7 Verwendung image 38
Listing 5.8 Verwendung create-styled-text 39
Listing 5.9 Verwendung render-output 40
Listing 5.10 Beispiel Clojure Funktionalität 41
Listing 5.11 Implementierung compose 44
Listing 5.12 Implementierung with-transform 45
Listing 5.13 Implementierung draw-fill 46
Listing 5.14 Implementierung polygon 47
Listing 5.15 Implementierung create-styled-text 48
Listing A.1 Composite Beispiel Quellcode 54
Listing A.2 AnyUp Open Graph Quellcode 58

1. Motivation

Das Unternehmen Effective Bodyweight Training GmbH (Geschäftsbezeichnung: AnyUp) ist ein Startup-Unternehmen, das eine mobile Applikation (App) für funktionelles Training mit dem eigenen Körpergewicht kostenlos zur Verfügung stellt. Mit dieser App ist es möglich Trainingseinheiten, sogenannte Workouts, zu absolvieren, die hierbei erzielten Ergebnisse zu speichern und sich mit anderen Benutzern auf einer Rangliste zu vergleichen. Die App ist für die Plattformen Android und iOS verfügbar. Das serverseitige Programm (Back-End) ist in der Programmiersprache Clojure geschrieben.

Das Unternehmen ist sehr jung und besitzt derzeit das primäre Ziel, seinen Bekanntheitsgrad zu erweitern. Zu diesem Zweck ist eine Erhöhung der Anzahl von aktiven Benutzer in der App von Bedeutung. Dieses Ziel soll über eine Erweiterung der Integration des sozialen Netzwerkes Facebook erreicht werden. Zurzeit ist es einem Benutzer möglich, sich in der App mit einem bestehendem Facebook-Konto anzumelden. Die Benutzer eines solchen Kontos sollen zukünftig die Möglichkeit besitzen, ihre erreichten Workout-Ergebnisse auf Facebook mit Freunden zu teilen. Für eine solche Mitteilungen muss ein Bild erstellt werden, auf dem Informationen zu Workout-Details und dem Benutzer zu lesen sind.

Als Mitarbeiter des Unternehmens AnyUp wurde mir aufgetragen, ein Programm zu erstellen, das die Komposition eines solchen Bildes ermöglicht. Nach vorangestellter Recherche stellte sich heraus, dass für Clojure derzeitig keine Anwendungen existieren, welche die Anforderungen des Unternehmens erfüllen. Diese Anforderungen können jedoch durch das Java 2D Application Programming Interface (API)[1], das auch in Clojure zugänglich ist, abgedeckt werden. Da diese Thematik ein großes Potential für ein Open Source Projekt birgt, das auch für andere Entwickler interessant sein könnte, fiel die Entscheidung auf die Entwicklung einer Domain Specific Language (DSL), zu Deutsch domänenspezifischen Sprache, in Clojure. Diese DSL wird die Features der Java 2D API nutzen, um Bilder zu komponieren. Dabei soll ein Bild nicht nur für den oben beschriebenen Anwendungsfall erstellt werden können, sondern generell bei Clojure-Projekten Verwendung finden, in denen Bildkompositionen erfolgen. Die Implementierung der DSL erfolgt in der Programmiersprache Clojure, da auch das Back-End des Unternehmens in dieser Sprache umgesetzt ist. Dies ist kein zwingender Grund für die Verwendung der Sprache. Da diese Sprache und die Idee hinter ihr jedoch ein großes persönliches Interesse bei mir weckten, lag eine intensive Analyse der Sprache Clojure im Rahmen dieser Arbeit nahe.

[1] API bezeichnet eine Schnittstelle zur Anbindung fremder Programme an ein Programm

2. Zielsetzung

Mit Hilfe der neu entwickelten DSL sollen Programme zur Komposition von Bildern schnell und unkompliziert entwickelt werden können. Bei der Entwicklung liegt der Fokus darauf, die Verwendung der 2D API zu erleichtern und Fehlerquellen während eines Entwicklungsprozesses mit der API zu vermindern. Dabei stellen sich die Fragen, ob sich die verwendete Programmiersprache Clojure für die Erstellung einer DSL eignet, und wieweit Multithreading durch den Einsatz einer funktionalen Programmiersprache erleichtert wird.

Für die Implementierung der DSL ist es notwendig, die Sprache Clojure mit ihrer generellen Funktionsweise sowie ihren grundlegenden Features näher zu betrachten. Diese Betrachtung findet im Abschnitt Grundlagen Clojure statt. Hierbei wird zunächst das funktionale Programmierparadigma beschrieben. Gleichzeitig werden Vergleiche zur objektorientierten (OO) Programmierung gezogen. Im Anschluss erfolgt eine Analyse der Sprache Clojure hinsichtlich der Syntax, den Basis-Datenstrukturen, sowie den grundlegenden Funktionalitäten wie die Java Interoperabilität oder das Makro-System. Der zweite Grundlagen-Teil dient dazu, dem Leser alle notwendigen Informationen bereitzustellen, die dieser zum Verständnis der entwickelten DSL mit dem Namen Image-Compojure benötigt. Hierzu zählt das Wissen, wodurch sich Begriff DSL definiert, was eine solche Sprache leisten kann, und wie die vorliegende DSL dieses auf Basis des Java 2D API ermöglicht. Nachdem alle Grundlagen erläutert sind, wird in dem Kapitel „DSL Image-Compojure“ die neu zu erstellende DSL betrachtet. Zu Beginn werden die Anforderungen definiert, die sich sowohl an die Features als auch an das Design der Sprache richten. In dem nachfolgenden Abschnitt wird erläutert, wie die neue Sprache verwendet werden muss. Jener Teil kann als Anleitung verstanden werden. Diese wird durch den Abschnitt Design und Implementierung noch weiter vertieft. Hier wird im Detail erläutert, wie die DSL konzeptioniert und programmiert wurde. Dabei stützt sich die Analyse auf die in Kapitel 3 und 4 gewonnenen Kenntnisse. Den Schluss des Kapitels bildet ein Ausblick auf das Programm, das unter Verwendung der DSL für die Firma AnyUp programmiert werden wird.

3. Grundlagen von Clojure

Clojure ist eine junge, auf LISP basierende Programmiersprache, die von Rich Heckey erstmals 2007 veröffentlicht wurde. LISP gehört zu einer der ältesten Sprachfamilien, die heute noch eingesetzt werden. Dabei bezeichnet LISP keine einzelne Sprache, sondern den Stil von Programmiersprachen. Worum es sich genau bei diesem Stil handelt, wird im Abschnitt 3.2 „Syntax“ näher betrachtet. Clojure fördert das funktionale Programmierparadigma, in dem es Funktionen als Werte behandelt. Sie ist auf der Java Virtual Maschine (JVM) gehostet und kann somit auf alle existierenden Java Bibliotheken zugreifen. Und zudem erleichtert Clojure durch die Verwendung von unveränderlichen (immutable), hoch performanten Datenstrukturen die Programmierung von Anwendungen, die Multithreading zur Optimierung von Laufzeiten implementieren. Durch den funktionalen Ansatz, die unveränderbaren Datenstrukturen und einer großen Anzahl von APIs um mit diesen zu arbeiten, ermöglicht Clojure das Erstellen von Programmen, die leichter zu testen und zu verstehen sind. (vgl. Clojure in Action, 2016, S. 1-2). Wodurch wird eine funktionale Sprache definiert? Dieser Frage widmet sich der nachfolgende Abschnitt.

3.1. Funktionale Programmierung

Die minimalste Anforderung an eine funktionale Programmiersprache ist, dass sie Funktionen als Werte behandelt. Dies bedeutet, dass Funktionen normale Werte wie der String „Image-Compojure“ oder die Zahl 42 sind. Funktionen können sowohl anderen Funktionen als Argumente übergeben werden als auch Funktionen als Rückgabewerte liefern. Zusätzlich besitzt ein Großteil der funktionalen Sprachen das Merkmale der Pure Functions mit referentieller Transparenz (Referential Transparency). Die Eigenschaft der referentiellen Transparenz sorgt dafür, dass Funktionen bei der Eingabe von denselben Argumenten jedes Mal dasselbe Ergebnis zurückliefern. Eine mathematische Rechnung wie 1 + 2 ist beispielsweise transparent, da diese immer das Ergebnis 3 liefern wird. Ein Gegenbeispiel stellt die Clojure-Funktion *rand* dar. Diese wird eine nicht vorhersehbare, zufällige Zahl als Ergebnis liefern und ist somit nicht transparent. Pure Functions liefern ein Ergebnis und haben keine nebenläufigen Effekte (Side Effects). Ein Beispiel für eine Funktion mit Side Effects ist *println*. Diese gibt *nil* zurück, gibt jedoch nebenläufig den übergebenen String in der Konsole aus. Der Schreibprozess beschreibt hier der Nebeneffekt (vgl. Clojure in Action, 2016, S. 3-4).

Standardmäßig unveränderbare Datenstrukturen stellen ebenfalls ein Feature dar, das in vielen funktionalen Sprachen implementiert ist. Dies bedeutet, dass Werte von Variablen, in Clojure Symbols genannt, sich nicht mehr verändern lassen. So wird es erleichtert, Pure Functions zu schreiben, da Argumente innerhalb von Funktionen nicht manipulierbar sind. Sprachen wie Clojure bieten Möglichkeiten an, den Wert eines

Symbols kontrolliert und explizit zu ändern. Abbildung 3.1 stellt die Werte eines Symbols „xs“ als Baum dar. Aus dem Wert des Symbols „xs“ kann eine neue Abstraktion der Werte zum Symbol „ys“ geschaffen werden, die viele Knoten und Blätter des ursprünglichen Baumes teilt und neue beinhaltet (blau dargestellt). Der Vorteil dieser Methode ist, dass die alten Werte des Baumes nicht überschrieben werden, sondern weiterhin existieren und verwendet werden können. Verändert man eine Datenstruktur in Clojure, wird diese nicht verändert, sondern eine Neue erstellt, in die bestehende Elemente kopiert werden. Ein anderes Beispiel: Subtrahiert man von dem Wert 42 den Wert 2, wird nicht der Wert 42 verändert, sondern zusätzlich der neue Wert 40 erstellt. Hierbei bilden Symbols Container, die zu unterschiedlichen Zeiten verschiedene Werte beinhalten können (vgl. Clojure in Action, 2016, S. 5).

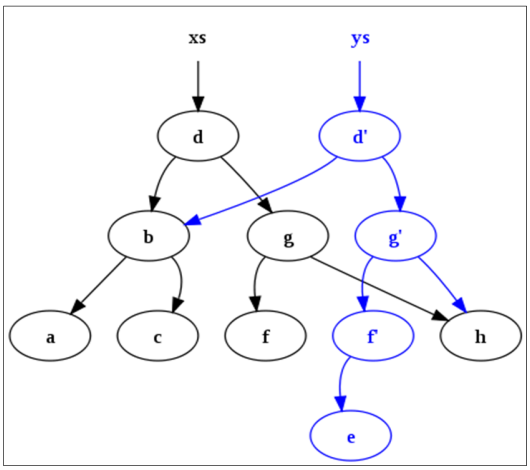

Abbildung 3.1 Symbol xs mit neuer Abstraktion ys (Quelle: Clojure in Action, 2016, S. 5)

3.1.1. State

In OO Sprachen ist es üblich, Objekte mit einem globalen Zustand (State) zu verwenden. Der Zustand eines Objekts wird durch seine Eigenschaften oder Attribute beschrieben. Diese Eigenschaften können von Funktionen verändert werden. Objekte sind somit veränderbar (mutable). In Java existiert ein Objekt, das zum Zeichnen eines Bildes verwendet wird. Dieses Objekt mit dem Namen Graphcis2D wird in dieser Arbeit noch eine zentrale Position einnehmen. Ein solches Objekt besitzt unter anderem die Eigenschaft Paint, welche die Farbe des aktuellen Zeichenstifts definiert. Ändert man die Farbe dieser Eigenschaft in eine andere Farbe, wird der Zustand des Objekts manipuliert.

„**State – you´re doing it wrong** “(Rich Heckey, Präsentation Boston Lisp Meeting, 2012)

Mit diesem Zitat behauptet Heckey, dass der Ansatz mit veränderbaren Objekten in Programmen nicht gut funktioniert und generell vermieden werden sollte. State macht es schwierig, Programme zu verstehen und stammt aus Zeiten, in denen Programme nicht mit parallellaufenden Threads arbeiten mussten. In Sprachen wie Java ist es durch die Verwendung von veränderbaren Objekten nicht unmöglich, jedoch höchst anspruchsvoll, Programme ohne Fehler bei der Verwendung von Multithreading-Features zu implementieren (vgl. Clojure in Action, 2016, S. 135).

3.1.2. Threads und das Problem State

Die Verwendung von globalen Objekten und deren Zuständen stellt unter bestimmten Vorrausetzungen kein Problem dar. Sobald Multithreading in einer Anwendung implementiert wird, kann es beim gleichzeitigen Zugriff auf globale Objekte zu ungewünschten Ergebnissen kommen. Ein Thread bezeichnet einen Teil eines Prozesses und stellt eine für sich eigenständige Aktivität zur Bewältigung einer Aufgabe in diesem Prozess dar (vgl. Internet: IT Wissen, Stand 23.05.2016, Threads o.S.). Der Begriff Multithreading beschreibt die Verwendung von multiplen Threads zur Bewältigung einer solchen Aufgabe. Hierzu werden die notwendigen Aktivitäten auf parallelläufige Threads aufgeteilt und müssen so nicht länger linear nacheinander abgearbeitet werden. Durch die parallele Bearbeitung wird die Laufzeit des Programms verbessert. Diese Nebenläufigkeit birgt jedoch einige Probleme, wie beispielsweise Updates, die bei einem gleichzeitig erfolgten Schreibprozess verlorengehen. Ein Beispiel: Thread A erhöht einen Zähler mit dem Wert 1 um den Wert 1, gleichzeitig erhöht Thread B den Wert ebenfalls. Das Ergebnis ist jedoch nicht 3, sondern 2. Thread B hat somit nicht erkannt, dass Thread A den Zähler bereits erhöht hat. Weitere Beispiele sind das Lesen von Daten, die bereits gelöscht wurden (phantom reads) oder das Lesen von Daten, die gerade verändert werden (dirty reads). Für diese Probleme gibt es in OO Sprachen Lösungen wie z.B. das Locking[2] (vgl. Clojure in Action, 2016, S. 136-137). Solche Lösungen steigern die Komplexität des Quellcodes. Clojure geht aus dem diesem Grund einen anderen Weg.

3.1.3. Lösungsweg in Clojure

Es bestehen die beiden Probleme der verlorenen Updates und des Lesens von inkonsistenten Daten. Letzteres behebt sich durch die Verwendung von unveränderlichen Datenstrukturen von allein. Durch die Tatsache, dass Daten nicht überschrieben oder gelöscht werden können, gibt es keine inkonsistenten Daten in Clojure. Ersteres Problem wird dadurch gelöst, dass Clojure den Zustand eines Symbols gänzlich anders manipuliert, als eine Sprache wie Java. OO Sprachen vermischen die Identität einer Variable mit dessen Wert. Dies bedeutet, dass man direkten Zugriff auf den Wert einer Variable im Speicher hat und diesen dort verändern kann. Wie in Abschnitt 3.1. erwähnt können Symbols in Clojure zu unterschiedlichen Zeiten andere Werte beinhalten. Symbols besitzen Referenzen (managed references), die auf unveränderliche Daten im Speicher verweisen. Anders als die Werte können diese Referenzen zur Laufzeit gerändert werden, wodurch ein Symbol auf einen anderen Wert zeigt. Durch die Unterscheidung von Identitäten und Werten ist es möglich, dass eine Sprache Semantik für nebenläufigen Threads (Concurrency Semantics) unterstützt.

[2] Sperrt Zugriff von anderen Threads, wenn bereits ein Thread auf eine Datenstruktur zugreift.

So sind z.B. Transaktionen möglich, die versuchen einen Wert zu erneuern. Schlägt dieser Prozess fehl, da ein anderer Prozess gleichzeitig die Zuweisung einer neuen Referenz vornimmt, wird der Vorgang im Anschluss wiederholt. Diese Thematik wird im Abschnitt 3.3 „Datenstrukturen“ erneut aufgegriffen (vgl. Clojure in Action, 2016 S.142-143).

3.2. Syntax

Die in Clojure verwendete Syntax ist auf die klassische Syntax einer LISP Sprache zurückzuführen. Ein Ausdruck wird von Klammern (…) eingeschlossen. Diese Klammern stellen die Datenstruktur List, zu Deutsch Liste, dar. Das erste Element eines validen Ausdrucks ist immer eine Funktion, ein Makro oder ein Special Form. Welche Bedeutung diese im Einzelnen haben, wird im späteren Verlauf betrachtet. Der Funktion folgt eine beliebige Anzahl von Argumenten. In Abbildung 3.2 sind zwei verschachtelte Ausdrücke (S-Expression) zu sehen. Clojure Programmcode ist somit eine Sammlung von Listen, die vom Compiler evaluiert werden. In diesem Fall liefert der Compiler das Ergebnis 11. Listen werden von innen nach außen verarbeitet. Zunächst wird somit die innere Liste (* 4 2) evaluiert. Der Operator für die Multiplikation (*) in OO Programmiersprachen ist in Clojure ebenfalls lediglich eine Funktion, die das Produkt der Argumente errechnet. In Clojure gibt es die übliche Schreibweise für mathematische Operationen wie 4 * 2 nicht. Clojure verwendet immer die Präfix-Notation: (* 1 2 3 4), die der Infix-Notation 1 * 2 * 3 * 4 in Java entspricht. Die innere Liste gibt den Wert 8 zurück. Dieser Wert tritt an die Stelle des zweiten Arguments der äußeren Liste. Somit enthält die Liste die Werte +, 3, und 8. Die Evaluation der Funktion (+) mit den Argumenten 3 und 8 gibt den Wert 11 zurück. An die Positionen der Zahlen können auch Funktionen treten, da diese bekanntlich Werte sind (vgl. Clojure in Action, 2016, S. 5). Ein Komma zum Separieren von Argumenten oder Werten ist generell nicht notwendig. Das Leerzeichen dient als Trennzeichen. Zur Übersichtlichkeit können Kommata verwendet werden, da Clojure diese ignoriert. Kommentare werden jeweils durch ein Semikolon (;) angegeben.

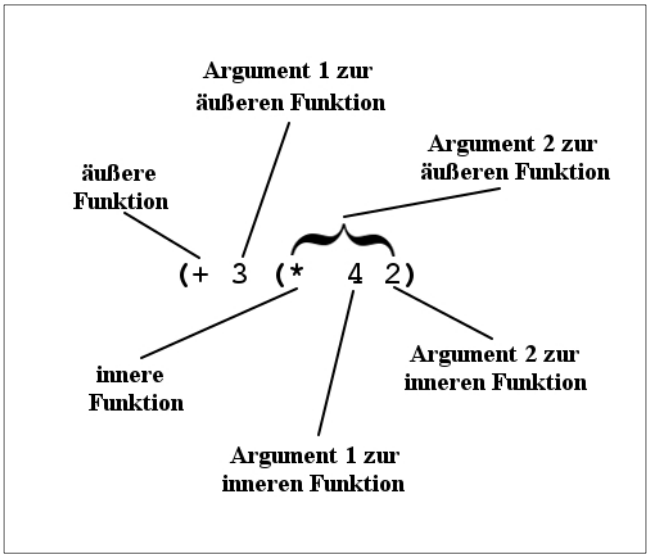

Abbildung 3.2 Verschachtelte Listen (angelehnt an: Clojure in Action, 2016 S. 10)

3.2.1. Special Forms

Clojure beinhaltet einige wenige sogenannte Special Forms (SF), die sich anders als Funktionen verhalten. Generell ist festzustellen, dass SF Grund-Funktionalitäten bereitstellen, die sich nicht mit Funktionen ermöglichen lassen. Zusätzlich können diese nicht wie Funktionen als Argumente an andere Funktionen übergeben werden. Ein weiterer Unterschied zu Funktionen ist der, dass SF nicht immer alle Argumente evaluieren müssen. Ein Beispiel hierfür ist *if*. Die If-Bedingung in Listing 3.1 dargestellt, evaluiert immer nur einen der beiden Ausdrücke. In diesem Fall wird immer der String „Hello“ zurückgegeben. Ist die Bedingung nicht *false* oder *nil*, wird der erste, andernfalls der zweite Ausdruck ausgeführt. An diese Stelle kann auch eine S-Expression treten (vgl. Clojure for the Brave and True, 2015, Clojure Alchemy: Reading, Evaluation, and Macros, Absatz: Special Form).

```
1(if false
2  (str "Hello" "World")
3  #(str "bye bye" "World"))
4 => Hello
```

Listing 3.1 If-Bedingung

In einer rein funktionalen Sprache ist dies ausreichend, da niemals noch weitere, nebenläufige Operationen erfolgen können. Solche Funktionen müssen keine Veränderungen an einem globalen State vornehmen. In realen Programmen sind Nebeneffekte jedoch zwingend notwendig. Beispiele stellen das Schreiben von Logs oder Konsolenausgaben dar. Um diese Nebeneffekte zu ermöglichen existiert das SF *do*. So ermöglicht das *do* Form multiple Ausdrücke nacheinander zu evaluieren, indem es die Ausdrücke in einer S-Expression zusammenführt (vgl. Clojure in Action, 2016, S. 37-38). Neben diesen beiden SFs gibt es noch weitere Anweisungen. Zu den alltäglichen Anweisungen zählen *loop*, *recur*, *let def*, und *quote*.

Durch die Forms *def* und *let* werden Werte einem Symbol zugewiesen (sog. Bindings). Der Unterschied zwischen den beiden ist, dass der Wert eines Symbols, welcher durch *def* zugewiesen wurde, global allen Funktionen im Namespace zur Verfügung steht. Ein Namespace bezeichnet einen Kontext (Scope), in dem eine Zuweisung zwischen einem Symbol und einem Var[3] gültig ist. Der Namespace *clojure.core* beinhaltet beispielsweise alle Kern-Funktionen, die einer Clojure-Anwendung zur Verfügung stehen. Im Gegensatz zu Def- sind Let-Bindings nur innerhalb des Scopes gültig, welcher durch die einschließenden Klammern des Ausdrucks bestimmt wird. Diese sind somit nur temporär verfügbar.

[3] Worum es sich bei einem Var handelt, wird noch erläutert. Zunächst reicht es zu wissen, dass es sich hierbei um eine Variable handelt.

Loop funktioniert analog zu *let*, jedoch mit dem Unterschied, dass *loop* den Einstiegspunkt für eine Rekursion, gestartet durch *recur* angibt. Eine Rekursion ist auch durch den Aufruf des eigenen Funktionsnamens möglich. Dies ist jedoch nicht empfehlenswert, da in Clojure keine Endrekursionsoptimierung vorhanden ist[4] und der direkte Aufruf somit den Speicher bei entsprechend häufiger Wiederholung an den Rand seiner Kapazität bringt. *Quote* ermöglicht es die Evaluation von Funktionen zu verhindern und stattdessen das Symbol selbst zurück zu geben. So gibt der Ausdruck *(quote (+ 1 2))* nicht den Wert 3 sondern die Liste *(+ 1 2)* zurück. Das spielt im Zusammenhang mit Makros eine zentrale Rolle.

3.2.2. Funktionen

Funktionen in Clojure sind Funktionen erster Klasse. Um dieser Definition gerecht zu werden, müssen Funktionen:

- ✓ Dynamisch erstellbar sein.
- ✓ Als Argumente übergeben werden können.
- ✓ Von Funktionen zurückgegeben werden können.
- ✓ Als Werte in anderen Datenstrukturen gespeichert werden können.

Um eine Funktion zu definieren, steht das Makro *defn* zur Verfügung. Dieses Makro ist eine Verbindung aus dem Makro *fn* und des SF *def*. Zusammenfassend lässt sich sagen, dass durch *fn* eine anonyme Funktion erstellt und durch *def* an ein Var gebunden wird. Wie eine Funktion definiert wird, ist in Listing 3.2 dargestellt. Direkt nach dem *defn* steht der Name des Symbols, an das die Funktion gebunden wird.

```
1 (defn adder
2   "Diese Funktion addiert die beiden Werte der Argumente a und b"
3   ([result & args]
4      (reduce (fn[result to-add]
5            (+ result to-add))
6             result
7             args))
8   ([b]
9    (adder 0 b)))
```

Listing 3.2 Funktionen definieren

Anschließend kann optional ein sogenannter Docstring angeführt werden, der im Regelfall für Erläuterungen zur Funktionsweise der Funktion dient. Dieser kann über die Funktion *doc* ausgegeben

[4] Endrekursions Optimierung ist nicht in der JVM realisiert

werden. Dem String folgen Argumente, die von eckigeren Klammern umschlossen werden. (s. Zeile 3 und 8). Funktionen ist es möglich, eine beliebige Anzahl an Argumenten (Arität) entgegen zu nehmen. Die Arität einer Funktion kann entweder fix definiert oder variabel sein. In dem vorliegenden Beispiel kann die Funktion mit einem Argument oder mit einer beliebigen Anzahl von Argumenten größer 1 aufgerufen werden. Je nachdem wie groß die Arität ist, wird der entsprechende Funktionskörper ausgeführt. Ist ein Argument angegeben, wird der Ausdruck in Zeile 9 ausgeführt. Dieser ruft die Funktion *adder*, also sich selbst rekursiv, mit zwei Argumenten auf. Auf diese Weise können Standardwerte, hier der Wert 0, beim Aufruf von Funktionen übergeben werden. Eine variable Arität ist in Zeile 3 gegeben. Dies wird durch das Zeichen (&) ermöglicht. Alle nachfolgenden Argumente werden zu einer Liste, mit dem Namen *args* hinzugefügt. Innerhalb des Funktionskörpers erfolgt eine Addition aller Argumente, deren Ergebnis zurückgegeben wird. Dies geschieht durch den Aufruf der Funktion *reduce*, einer Funktion höherer Ordnung, die im Verlauf dieses Kapitels noch einmal näher betrachtet werden wird. Das Makro *fn* erstellt eine anonyme Funktion, die zwei Argumenten besitzt und die Addition selbst ausführt. Anonyme Funktionen heißen anonym, da sie keinen Namen besitzen und nur für diesen kurzen Zeitraum zur Verfügung stehen. Dieser Funktions-Typ kann auch über das Zeichen # erstellt werden (s. Listing 3.1 S.7 Z. 2).

3.3. Datenstrukturen

In diesem Abschnitt werden die Datenstrukturen aus dem Kern-Namespace *clojure.core* betrachtet. Zunächst einige erläuternde Worte zu den grundlegenden Strukturen:

Der Wert *nil* entspricht dem Wert *Null* in Java. Alle Werte bis auf *nil* und *false* werden als *true* interpretiert. Characters in Clojure entsprechen ebenfalls Java Characters. Strings werden in doppelten Anführungszeichen (“) angegeben. Das einfache Zeichen (‘) stellt in Clojure ein Makro dar und wird im späteren Verlauf betrachtet. Standardmäßig wird in Clojure der Datentyp *long* für ganze Zahlen und *double* für Fließkommazahlen verwendet. Für größere Zahlenräume stehen *bigint* oder *bigdec* zur Verfügung.

3.3.1. Symbols und Keywords

Bisher wurden Symbols wie Variablen in OO Sprachen interpretiert. Dieser Vergleich ist nicht gänzlich korrekt. Symbols besitzen Referenzen bzw. Verweise auf den Wert einer Variablen, z.B. des Typs Var, im Speicher. Ein Symbol bezeichnet den Namen der derzeitigen Referenz, welche auch selbst als Wert verwendet werden kann. Dies liegt daran, dass Clojure eine Unterscheidung zwischen Lesen und Evaluieren des Programmcodes macht. Eine Thematik, die im Zusammenhang mit Makros deutlich werden wird (vgl.

Kapitel 3.6). Symbols können mit allen Buchstaben oder den Zeichen *, !, _, ?, $, %, &, =, <, > benannt werden. Beschränkungen bestehen darin, dass der erste Buchstabe keine Zahl sein darf. Ist das erste Zeichen ein (-), (+) oder (.) darf auch der zweite Buchstabe keine Zahl sein (vgl. Clojure in Action, 2016, S. 26). Wird ein Symbol verwendet, ohne das diesem ein Wert zugewiesen wurde, kommt es zu einer Compiler Exception, da das Symbol nicht aufgelöst werden kann. Nutzt man *quote* oder das Makro (‘), wird der Name des Symbols selbst zurückgegeben und kein Fehler tritt auf. Keys sind Symbols sehr ähnlich. Der Name eines Keys beginnt immer mit einem (:). Keys sind Funktionen, die beim Zugriff auf Werte in Maps und Sets verwendet werden.

3.3.2. List

Eine Liste stellt die Basis-Datenstruktur für Sammlungen (Collections) von Werten in Clojure dar. Bei einer solchen Liste handelt es sich um eine einfach verkettete Liste (singly linked list), durch die es möglich ist, Elemente von vorne nach hinten, jedoch nicht von hinten nach vorne abzufragen. Listen können durch die Funktion *list* erstellt werden. Ein Erstellen ist ebenfalls durch das Makro (‘) möglich.

```
1 (def test1 (list a 2 3))
2 (def test2 (a 2 3))
3 (def test3 '(a 2 3))
```

Listing 3.3 Lists

In Listing 3.3 werden drei Symbols mit den Namen test1-3 definiert. In Zeile 1 wird hierzu die Funktion *list* benutzt. In Zeile 3 wird zum Erstellen der Liste das Makro (‘) verwendet. Im Abschnitt Syntax wurde bereits erwähnt, dass valider Clojure-Quellcode verschachtelte Listen des hier beschriebenen Typs darstellen, die evaluiert werden. Clojure wird folglich durch seine eigene Datenstruktur ausgedrückt und besitzt somit die Eigenschaft der Homoikonizität.

Der Aufruf in Zeile 2 wird zu einem Fehler führen, obwohl es sich dabei um eine valide Liste handelt. Das erste Element dieser Liste ist ein Symbol mit dem Namen *a*. Grund für die Fehlermeldung ist, dass es sich zwar um eine Liste jedoch nicht um einen validen Ausdruck handelt. Der Compiler erwartet an der Stelle des Symbols *a* eine Funktion, SF oder ein Makro. Da *a* jedoch zuvor nicht definiert wurde, kann die Auflösung nicht erfolgen. Anders verhält es sich in Zeile 3. Durch das Makro (‘) wird dem Compiler mitgeteilt, dass an dieser Stelle die Liste nicht evaluiert, sondern lediglich gelesen werden soll. Es wird also nur die Liste mit dem Namen des Symbols zurückgegeben.

3.3.3. Vector

Ein Vector, zu Deutsch Vektor, verhält sich wie eine Liste, mit dem Unterschied, dass dieser durch eckige Klammern [ ] beschrieben wird und Werte einen nummerischen Index besitzen. Vektoren können durch die Funktion *vector* oder direkt über die Angabe von Klammern erstellt werden. Die Zeilen 1 und 2, dargestellt in Listing 3.4, erstellen beide einen Vektor. Die dritte Zeile wird das Element mit dem Index 0 aus der angegeben Collection ausgeben, in diesem Fall den Wert 1. Eine weitere Eigenschaft von Vektoren ist, dass diese auch Funktionen sind, welche ein Argument entgegennehmen können.

```
1 (vector 1 2 3)
2 [1 2 3]
3 (nth [1 2 3] 0)
```

Listing 3.4 Vektoren

Neben der oben dargestellten *nth* Funktion kann ein Zugriff auf einen Index auch in der Form *([1 2 3] 0)* erfolgen. Dieser Ausdruck liefert dasselbe Ergebnis wie Zeile 3 (vgl. Clojure in Action, 2016, S. 29-30). Analog zu Vektoren funktionieren Sets. Diese können gleiche Werte jedoch nicht mehrfach enthalten (unique).

3.3.4. Map

Maps sind in Clojure das, was in anderen Programmiersprachen wie Ruby oder Python assoziative Arrays oder Wörterbücher sind. Diese werden durch die Funktion *hash-map* oder durch geschweifte Klammern { } definiert. Hierbei müssen die Argumente immer aus Key-Value-Paaren bestehen. Ein Wert (Value) in einer Map besitzt immer einen Schlüssel (Key) und umgekehrt. Auf einen Value einer Map kann durch seinen Key zugriffen werden. Neben dem normalen Zugriff auf Values können auch Standardwerte angegeben werden, sofern ein Key beim Zugriff auf eine Map nicht vorhanden ist. Da diese Keys Funktionen darstellen, sind Aufrufe wie in Zeile 2 und 3 möglich (siehe Listing 3.5). Zeile 1 gibt eine Map mit den Key-Value-Paaren *:a 1* und *:b 2* zurück. Zeile 2 liefert den Wert 1 zu dem Key :*a* aus der Map. Da in Zeile 3 der Key *:c* nicht in der Map vorhanden ist, wird der Wert 3, der dem Key als zweites Argument übergeben wurde, zurückgeliefert (vgl. Clojure in Action, 20116, S. 31).

```
1 {:a 1 :b 2}
2 (:a {:a 1 :b 2})
3 (:c {:a 1 :b 2} 3)
```

Listing 3.5 Maps

Gewünschte Werte aus Maps, Sets und Vektoren können an Symbols durch ein Verfahren mit dem Namen Destructuring gebunden werden. Dieses findet oft in der Liste der Argumente einer Funktion Verwendung. Listing 3.6 zeigt die Verwendung von Destructuring bei der Übergabe einer Map an eine Funktion, wodurch die Values der Keys aus der Map an gleichnamige Symbols gebunden werden.

```
1 (defn greet-user [{:keys [first-name last-name]}]
2   (println (str "welcome " first-name " " last-name)))
3
4 (greet-user {:first-name "Nicolas" :last-name "Schwartau"})
5
6 => Welcome Nicolas Schwartau
```

Listing 3.6 Destructuring (angelehnt an Clojure in Action, 2016, S. 95)

3.3.5. Sequence

Der Begriff Sequence beschreibt keine eigene Datenstruktur, sondern ein Interface. Dieses Interface definiert eine Abstraktion einer Collection, dessen Elemente in einer linearen Reihenfolge angeordnet sind. So können alle Datenstrukturen, die dieses Interface implementieren, analog zur Datenstruktur List behandelt werden. Auf diese Weise wird die Verwendung von Funktionen wie *first*, die das erste Element, oder *rest*, die alle nachfolgenden Elemente einer Sequence ausgibt, ermöglicht. Tatsächlich implementieren alle bisher beschriebenen Datenstrukturen dieses Interface. So müssen Funktionen lediglich im Kontext dieser Abstraktion definiert werden, um mit allen bisher beschrieben Datenstrukturen zu funktionieren (vgl. Clojure for the Brave and True, 2015, Kapitel: Core Functions in Depth, Absatz: Programming to Abstractions).

3.3.6. Persistent Datastructure

Hinter diesem Begriff verbirgt sich ebenfalls keine neue Datenstruktur, sondern ein Prinzip, auf dem sich alle Kern-Datenstrukturen in Clojure stützen. Eine Datenstruktur ist persistent, wenn sie nach einer Änderung ihren vorherigen Stand erhält (vgl. Kapitel 3.1). Da die Daten in Clojure immutable sind und nur neue Referenzen angegeben werden können, sind alle vorherigen Datenstände weiterhin existent und können wieder hervorgebracht werden. Ein Beispiel: Wird eine neue Bindung an ein Symbol durch einen Let-Block vorgenommen, verweist das Symbol nach Verlassen des Let-Blocks wieder auf den vorherigen Wert (vgl. Clojure in Action, 2016 S. 144).

3.3.7. Managed Reference

Um unveränderbaren Datenstrukturen neue Werte zuweisen zu können, stehen in Clojure die Managed References zur Verfügung. Durch die Verwendung des SF *def* wird eine Referenz des Typs Var erstellt. Diese ist zunächst fix definiert. Wird der Zusatz ^*:dynamic* verwendet, kann über die Funktion *binding* ein neuer Wert hinzugefügt werden. Listing 3.7 zeigt wie das funktioniert. Zunächst wird der String *„standard“* an das Symbol *standard* gebunden. Bei der Definition des zweiten Symbols *new-binding* wird die Referenz geändert und zeigt auf den String *„neu“*. Das Symbol *new-binding* gibt nun den String *„neu“* anstatt *„standard“* zurück. Es existieren noch die drei weiteren Managed References Typen ref, agent und atom, die im Rahmen dieser Arbeit jedoch keine Rolle spielen. Welcher Typ verwendet werden sollte, ist von dem vorliegenden Fall abhängig. Eine Referenz des Typs Var kann bei Fällen angewendet werden, in denen es für Threads nicht relevant ist, ob ein anderer Thread gleichzeitig auf den Wert eines Symbols zugreift.

```
1 (def ^:dynamic standard "standard")
2
3 (def new-binding
4   (binding [standard "neu"]
5         standard))
```

Listing 3.7 Var Binding

3.4. Funktionen höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, die entweder Funktionen als Argumente entgegennehmen oder eine Funktion als Rückgabewert liefern. Diese Art von Funktionen werden meist dann eingesetzt, wenn sich der Quellecode andernfalls wiederholen würde. Bestandteil des Clojure Kern-Namespaces sind unter anderem die beiden Funktionen *map* und *reduce*, deren Verwendung im Folgenden exemplarisch skizziert wird. Diese beiden Funktionen gehören zu den am häufigsten verwendeten Werkzeugen in der Clojure Entwicklung, da Schleifen, wie sie OO Sprachen üblich sind, nicht existieren.

Map nimmt eine Funktion und eine Sequence entgegen. Sie wendet die Funktion auf alle Elemente an, die sich in der Collection befinden. Diese liefert im Anschluss eine neue Sequence, welche die neuen Elemente beinhaltet. Der Ausdruck *(map inc [1 2 3])* liefert das Ergebnis *(2 3 4)* zurück. *Reduce* liefert bei dem Beispiel *(reduce * 1 [2 21])* keine Sequence, sondern den Wert *42*. Hierbei wird die übergebene Funktion auf die nächsten beiden Elemente der Collection angewendet, bis ihr Ende erreicht ist. Ist ein Startwert angegeben, in diesem Fall 1, erfolgt die erste Zuweisung auf den Startwert sowie auf das erste Element der Collection. In beiden Fällen erspart der Entwickler sich Schleifenaufrufe.

3.5. Java Interoperabilität

Java und Clojure verwenden beide die Java Virtual Machine (JVM). Das liegt unter anderem an der Tatsache, dass ein Großteil der Sprache Clojure in Java implementiert ist. Diesbezüglich versucht Clojure den Ursprung nicht zu verbergen, sondern unterstützt aktiv die Verwendung von Java und bietet eine nahezu nahtlose Interoperabilität (Interop) zwischen beiden Sprachen an. So kann für den Fall, dass in Clojure Funktionalitäten nicht realisiert sind, immer auf eine bestehende Java Bibliothek zurückgegriffen werden. Jedes Programm, das Java als Sprache benötigt, kann ebenfalls in Clojure realisiert werden. Es ist möglich Instanzen von Klassen zu erstellen und auf Funktionen und Variablen von diesen zuzugreifen. Sogar die Einbindung von Interfaces und Erweiterung von Klassen sind möglich. Im Folgenden wird kurz auf die grundlegende Anwendung des Interops und zwei Makros eingegangen, die in Verbindung mit dem diesem verwendet werden können.

Instanzen werden durch das SF *new* erstellt. Anstelle von *new* steht auch die Verwendung des Makros (.) zur Verfügung. Beispielsweise kann eine Instanz der Klasse SimpleDateFormat durch den Ausdruck *(new SimpleDateFormat „yyyy-MM-dd“)* oder *(SimpleDateFormat. „yyyy-MM-dd“)* erstellt werden. Das SF *new* erwartet einen Klassennamen und Argumente des Konstruktors einer Klasse. Im Listing 3.8 ist ein Beispiel für die Erstellung einer Instanz der Klasse JFrame mit einer überschriebenen *paint* Methode zu sehen. Diese Instanz wird durch das Makro *proxy* erzeugt. Der Klassen- und der Methodenname wird dem Makro zusammen mit der neuen Methoden-Logik übergeben (siehe Z. 1-3). In Zeile 3 erfolgt der Aufruf der Methode *drawimage* aus einer Instanz *g* der Klasse Graphics. Zugriffe auf Methoden und Variablen von Klasseninstanzen oder Klassen erfolgen nach dem Schema: *(.Name Klasse args*)*. Der Zugriff auf statische Methoden erfolgt auf eine andere Art: *(Klasse/Name)* Name steht hier für den Namen einer Variable oder einer Methode (vgl. Clojure in Action, 2016, S 118 -119).

```
1(let [frame (proxy [JFrame] []
2              (paint [#^Graphics g]
3               (.drawImage g image 0 0 nil)))]
5  (doto frame
6    (.setSize dimensison)
     (.setVisible true))
```

Listing 3.8 Interop Beispiel

Clojure bietet eine Reihe von Makros, um das Arbeiten mit Java Objekten zu erleichtern. Ein Beispiel hierfür ist *doto*. *Doto* ruft alle nachfolgenden Methoden und Variablen der angegebenen Instanz ab. An dieser Stelle wird *doto* verwendet um das erzeugte JFrame Objekt eine Größe zuzuweisen und es anschließend sichtbar zu machen.

3.6. Makro-System

Der Grund weshalb Clojure homoikonisch konzipiert wurde, ist das Ermöglichen des Makro-Systems. Der Makro-Ansatz ist idiomatisch, denn ein Großteil der in Clojure zur Verfügung stehenden Features wurde durch Makros implementiert. Makros können Programme schreiben, die Programme schreiben. Diese Art der Programmerzeugung wird auch als Meta-Programming bezeichnet. Die Laufzeit in Clojure wird in zwei

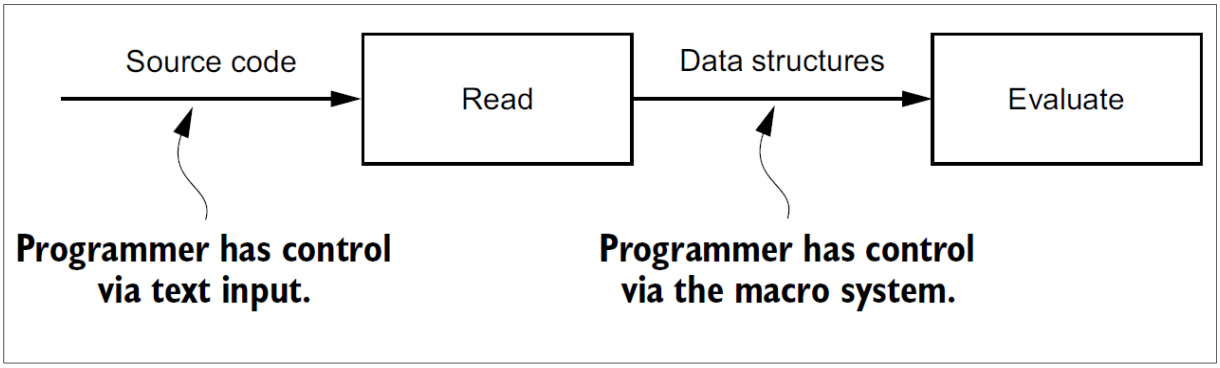

Abbildung 3.3 Phasen zur Clojure Laufzeit (Quelle: Clojure in Action, 2016, S. 167)

Phasen aufgeteilt. Die erste Phase beschreibt das Lesen des Quellcodes, die zweite wird durch die Konvertierung des Quellcodes in die Datenstruktur List und deren anschließende Evaluation dargestellt. Abbildung 3.3 skizziert, an welchen Stellen ein Programmierer die Möglichkeit hat, in die Phasen einzugreifen. Anders als in den meisten Sprachen kann der Entwickler nach dem Lesen des Quellcodes noch Änderungen an den erstellten Datenstrukturen vornehmen, bevor diese evaluiert werden. Der Eingriff erfolgt durch die Verwendung des Makro-Systems. Makros sind Funktionen, welche durch die Bearbeitung von Datenstrukturen, eine programmatische Modifizierung des Quellcodes erzielen. Sie ermöglichen die Implementierung einer eigenen Syntax und neuer Features (vgl. Clojure in Action, 2016, S. 166 -167).

Die Definition eines Makros entspricht der einer Funktion. Diese wird jedoch nicht durch *defn*, sondern *defmacro* angegeben. Intern sind Makros äquivalent zu Funktionen, diese sind jedoch durch Metadaten als Makros gekennzeichnet[5]. Der Unterschied zwischen beiden ist, dass Funktionen zum Erhalt von Werten und Makros zum Erhalt von S-Expressions (Vgl. 3.2.1) ausgeführt werden. Die S-Expressions werden anschließend wiederum evaluiert. Makros operieren nur mit Namen von Symbols und haben keinen Zugriff auf deren Werte. Was dies bedeutet, wird anhand des Beispiels in Listing 3.9 erläutert.

[5] Metadaten enthalten Daten zu Daten. So kann zwischen unterschiedlichen Identitäten bei gleichen Datenstrukturen unterschieden werden. Abhängig von der Identität können unterschiedliches Verhalten erzielt werden. Metadaten sind üblich in Clojure, jedoch nicht weiter für die Arbeit relevant.

Dieses Beispiel ermöglicht die Verwendung der infixen Schreibweise für Berechnungen, wie sie in Sprachen wie Java üblich ist. Zunächst erfolgt die Definition des Makros *infix*. Dieses nimmt ein Argument *expr* entgegen, bei dem es sich um einen Ausdruck bzw. Liste handeln muss. Durch Destructuring werden die Elemente des Arguments *expr* an die Symbols *left op right* übergeben. Die Funktion *list* in Zeile 3 erstellt eine neue Liste, in der die Elemente so angeordnet sind, dass diese evaluiert werden können (siehe Z. 8). So kann der Ausdruck in Zeile 5 mit der gewohnten Notion berechnet werden. Durch die Funktion *macroexpand* kann der vom Makro zurückgelieferte Ausdruck ausgegeben werden, bevor dieser evaluiert wird. In diesem Fall handelt es sich um validen Clojure Code (siehe Z. 8).

```
1 (defmacro infix [expr]
2   (let [[left op right] expr]
3     (list op left right)))
4
5 (infix (1 + 1))
6  => 2
7 (macroexpand '(infix (1 + 2)))
8  => (+ 1 2)
```

Listing 3.9 Makro Beispiel (angelehnt an Clojure in Action, 2016, S. 180)

In dem Fall, dass es sich bei *infix* nicht um ein Makro, sondern um eine Funktion handelt, kommt es an dieser Stelle zu einem Fehler. Ausdrücke werden immer evaluiert, sobald diese übergeben werden. Bei der Evaluation wird aufgrund der falschen Reihenfolge der Elemente im Ausdruck versucht, den Wert 1 als Symbol aufzulösen, obwohl dieser nicht existiert. Alle Funktionen folgen den beiden Regeln:

1. Evaluiere alle Argumente, die an die Funktion übergeben wurden.
2. Evaluiere die Funktion mit den Werten der Argumente.

Sofern eine Regel nicht eingehalten werden darf, muss ein Makro oder eine Closure verwendet werden (vgl. Clojure in Action, 2016, S. 169).

3.7. Closures

Closures sind ein zentrales Feature von funktionalen Programmiersprachen. Dieses Feature ist von solch zentraler Bedeutung, dass selbst der Name Clojure von Closures abgeleitet wurde. Closures sind Funktionen, die auf sogenannte freie Variablen (Free Variables) außerhalb ihres eigenen Geltungsbereichs (Scope) zugreifen können. Freie Variablen bezeichnen Variablen, die weder als Argument noch als lokale Variable einer Funktion zur Verfügung stehen. Diese können beispielsweise über *let* erstellt werden, so geschehen in Listing 3.10. Die Funktion *concat-hello* erstellt eine Variable *greeting*. Diese wird von der anonymen Funktion verwendet (siehe Z. 3). *Concat-hello* gibt die anonyme Funktion zurück. Da *greeting* außerhalb des Scopes der anonymen Funktion liegt, handelt es sich bei der anonymen Funktion um eine Closure. Die Lebensdauer der Variable *greeting* wird auf die Lebensdauer der zurückgelieferten Closure ausgedehnt. An das Symbol *message* wird in Zeile 7 die Closure mit der Begrüßungsformel „Hello Nicolas“ als String gebunden. Der Aufruf von *message* mit einem String als Argument evaluiert die Closure letztendlich und gibt die gesamte Nachricht auf der Konsole aus. Bei diesem Beispiel wurde die Konsolenausgabe verzögert. Es ist ebenfalls denkbar, dass beispielsweise bei einer *If*-Bedingung, Closures niemals tatsächlich evaluiert werden. Dies kann wünschenswert sein, wenn Funktionen anderen Funktionen als Argumente übergeben werden. Eine solche Verzögerung wäre anderweitig nur mit Makros möglich.

```
1 (defn concat-hello[name]
2   (let [greeting (str "Hello " name ", ")]
3        #(println (str greeting %))))
4
5 (def message (concat-hello "Nicolas"))
6
7 (message "Are you done yet?")
8
9 => Hello Nicolas, Are you done yet?
```

Listing 3.10 Closures

4. Grundlagen der DSL

Dieses Kapitel beschäftigt sich mit den Grundlagen, die für die Erstellung der domänenspezifischen Sprache Image-Compojure benötigt werden. Hierzu wird zunächst der Begriff DSL definiert. Anschließend wird beschrieben, worum es sich bei der Java 2D API handelt und wie diese generell funktioniert. Dabei werden insbesondere die Features der API beschrieben, welche im späteren Verlauf der Arbeit Anwendung finden.

4.1. Domain Specific Language

Eine DSL ist eine Sprache[6], die nach Martin Fowler über mehrere Schlüsselelemente definiert wird. Zum einen muss eine DSL ein so genanntes Fluent Interface implementieren. Das bedeutet, dass einzelne kurze Ausdrücke zu längeren Ausdrücken zusammengesetzt werden können, wodurch sich der Quellcode fließend lesen lässt. Ein Beispiel hierfür ist das Erstellen neuer oder manipulierter Objekte durch Methodenketten, das sogenannte Method Chaining. Hierbei werden Methoden nacheinander auf ein initial angegebenes Objekt angewendet. Diese Methoden geben dabei das durch sie selbst manipulierte Objekt an die nächste Methode weiter (vgl. Domain Specific Language, 2010, S. 32).

Des Weiteren wird eine DSL über seine limitierte Ausdrucksfähigkeit definiert. Eine General Purpose Language[7] (GPL) bietet ein breiteres Feld an Möglichkeiten. Diese Vielfalt macht es für einen Entwickler jedoch schwieriger, die GPL in ihrem vollen Umfang zu erlernen und anzuwenden. Eine DSL bietet nur eine begrenzte Anzahl an Features. Somit können keine vollständigen Anwendungen erstellt, sondern lediglich einzelne Komponenten einer Anwendung hinzufügt werden. (vgl. Domain Specific Language, 2010, S. 32)

GPLs und DSLs schaffen einem Programmierer die Möglichkeit, Computer zu instruieren. Hierbei besitzen alle GPLs die Eigenschaft der Turing-Vollständigkeit. Sie sind somit universell einsetzbar, wodurch ein Programm in jeder vorhandenen Programmiersprache wie Java, Clojure, Ruby oder Anderen entwickelt werden kann. Hierbei ist anzumerken, dass eine DSL die Turing-Vollständigkeit nicht mehr besitzt. Wächst die DSL durch Weiterentwicklung bis zur Turing-Vollständigkeit, spricht man wieder von einer GPL.

Es stellt sich daher die Frage, warum es überhaupt unterschiedliche Programmiersprachen gibt und nicht alle Programmierer dieselbe Sprache nutzen. Eine Erklärung hierfür ist, dass Programmiersprachen für ihren

[6] Generell ist eine Programmiersprache gemeint. DSLs können jedoch auch Programme beschreiben, die ihrerseits Programme schreiben. Eine Shell-Programm, durch das Quellcodedateien erzeugt werden können, ist in diesem Sinne auch eine Sprache.
[7] Bezeichnet Programmiersprachen, die für alle Anwendungsfälle verwendet werden können, wie z.B. Java.

jeweiligen Einsatzzweck optimiert worden sind. So wird zum Beispiel die Sprache C hauptsächlich zur Entwicklung von Betriebssystemen wie Linux-Distributionen oder eingebetteten Systemen[8] verwendet. Ein Grund dafür ist die Hardwarenähe der Sprache. So wird es erleichtert Speicher direkt zu adressieren, ein wichtiger Aspekt bei der Kommunikation zwischen Zentraleinheit und Peripheriegeräten. Auch können Zeiger[9] genutzt werden, die es ermöglichen, effiziente Datenstrukturen zu konstruieren. (vgl. DSL Engineering, S. 27-28; Domain Specific Language, 2010, S.33)

Clojure hingegen biete durch Closures die Möglichkeit, die Evaluation von Funktionen auf einen späteren Zeitpunkt zu verschieben oder gar Informationen vor einem Aufrufer einer Funktion zu verbergen, obwohl diese Informationen für die Funktion selbst sichtbar sind. Dies sind Features, die bei Webapplikationen sehr hilfreich sein können. Ein weiterer Vorteil ist die Verwendung der weit verbreiteten JVM. Mit dieser ist es möglich, Clojure Programme auf jedem Rechner anzuwenden, auf denen diese virtuelle Maschine installiert ist. Nicht zuletzt erleichtert es Clojure durch die Eigenschaft der Homoikonizität und des idiomatischen Makro-Systems, mit verhältnismäßig wenig Aufwand komplexe interne DSLs zu implementieren. Eine Eigenschaft, die sich in Java beispielweise weitaus schwieriger umsetzten lässt, da in dort die Beschreibung einer eigenen Syntax nur bei der Definition von Klassen möglich ist (vgl. Internet: innoQ, Stand 05.05.2016, o.S.). Ziel einer DSL ist es, durch die Optimierung für einen bestimmten Einsatzzweck die Möglichkeit einer schnelleren und effizienteren Softwareentwicklung zu gewähren. Gleiche Ergebnisse können mit deutlich weniger Quellcode erzielt werden, welcher zudem einfacher zu verstehen und somit auch besser zu warten sein wird.

Aus dem Punkt der limitierten Ausdrucksfähigkeit resultiert das nächste Element: Die spezifische Domäne. Dieses Element ist auch gleichzeitig der Namensgeber des Begriffs DSL. Eine DSL wird für einen spezifischen Anwendungsbereich bzw. Domäne definiert. (vgl. Domain Specific Language, 2010, S. 32) Im Falle dieser Arbeit wird die Domäne durch die Komposition eines Bildes mit Hilfe der Java 2D API beschrieben.

Man unterscheidet zwischen internen und externen DSLs. Martin Fowler beschreibt eine externe DSL als eine Sprache, die von der Hauptsprache des Programms separiert ist (vgl. Domain Specific Language, 2010, S. 32). Die beiden Sprachen können sich eine gemeinsame Syntax teilen, oft unterscheiden sich diese jedoch grundlegend. Als Beispiel kann hier XML angeführt werden. Im Regelfall werden Skripte, die in einer

[8] Eingebettete Systeme sind in einem technischen Kontext eingebundene Systeme. Sie übernehmen die Steuerung, Regelung und Überwachung eines Systems

[9] Adressierung eines Wertes einer Variable nicht über deren Namen, sondern über die direkte Adresse im Speicher.

externen DSL geschrieben werden, durch Programmcode in der Hauptapplikation geparsed[10]. Externe DSLs werden oft verwendet, ohne das Kenntnis darüber herrscht, dass es sich um eine DSL handelt. So sind zum Beispiel reguläre Ausrücke, eine Möglichkeit zur Beschreibung von Mengen in Strings, oder auch SQL[11], eine Sprache zum Arbeiten mit relationalen Datenbanken, alltägliche Werkzeuge und externe DSLs.

Eine interne DSL beschreibt Martin Fowler hingegen als eine Sprache, die eine GPL in einer bestimmten Art und Weise verwendet (vgl. Domain Specific Language, 2010, S. 32). Diese DSL wird in der Sprache der übergeordneten GPL geschrieben. Sie stellt eine Teilmenge der Mechanismen der GPL für einen bestimmten Zweck in einem eigens definierten Stil bereit. Für eine interne DSL muss kein Compiler oder Interpreter entwickelt werden, da sie auf die Mechanismen der Hostsprache zugreifen kann. Ein Indiz für eine gute DSL ist es, wenn sich bei der Verwendung stets das Bewusstsein einstellt, dass man sich im Kontext der DSL und nicht in dem der Hostsprache befindet. So können Fehler im Zusammenhang mit Syntaxbesonderheiten vermieden werden.

Ein Beispiel für eine GPL mit vielen DSLs stellt Ruby dar. So ist das Web-Framework Rails eine Sammlung von vielen internen DSLs und APIs. Wo verläuft hier jedoch die Grenze zwischen einer DSL und einer API? Es gibt keine klare Grenze, die man ziehen kann. Vielmehr ist der Übergang zwischen ihnen fließend. Fowlers Ansatz ist hierzu der, dass eine Bibliothek oder API einen Wortschatz zur Verfügung stellt, eine DSL hingegen die Grammatik mitliefert. Die Möglichkeit ganze Sätze aus voneinander getrennten Befehlen zusammenzusetzen, macht aus einer API ein Fluent Interface und somit eine interne DSL (vgl. Domain Specific Language, 2010, S. 32 – 33). Ein Beispiel aus dem Rails-Framework ist der darin enthaltene Mechanismus zur Definition von Beziehungen zwischen Modell-Entitäten[12], dargestellt in Listing 4.1. Hinter den Keys, beispielweise belongs_to, verbergen sich Funktionen, die erst zur Laufzeit den Quellcode für die Unterstützung der Assoziationen erzeugen. Man beachte, dass sich das Listing wie ein Text lesen lässt, der nicht nur von Entwicklern verstanden werden kann (vgl. Internet: innoQ, Stand 10.05.2016).

```
1 class Office < ActiveRecord::Base
2     belongs_to          :region
3     has_one             :manager
4     has_many            :employees
5 end
```

Listing 4.1 Assoziationen in Rails

Ebenso ist auch die entwickelte DSL Image-Compojure, auf der diese Arbeit basiert, eine interne DSL. Die Implementierung erfolgte durch die Verwendung von validem Clojure-Code. Wobei für diese DSL

[10] Parsen beschreibt einen Prozess zur Zerlegung und Umwandlung von Code oder einer Einagabe in ein Format, das von der erhaltenden Anwendung interpretiert werden kann.
[11] SQL steht für Structured Query Language
[12] Assoziationen beschreiben Beziehungen: z.B. Besitzt ein Objekt der Klasse A ein oder n Objekte der Klasse B

theoretisch zwei GPLs als Hostsprache dienen. Da Clojure und Java dieselbe JVM nutzen und Clojure durch die Interoperabilität nativ Java spricht und umgekehrt, ist dies an dieser Stelle nicht weiter relevant.

4.2. Java 2D API

Die Java 2D API ermöglicht es dem Entwickler zweidimensionale Bilder durch Erweiterung des Abstract Windowing Toolkit [13] zu erstellen. Hierbei existieren 2D Objekte in einem zweidimensionalen Koordinatensystem. Der Ursprung des Koordinatensystems entspringt in der oberen linken Ecke des User Space. Als User Space wird das Koordinatensystem bezeichnet, in welchem grafische Primitive definiert werden. Steigende X-Werte verlaufen nach rechts, steigende Y-Werte nach unten. Koordinaten werden mit Integer Werten angegeben. Fließkommazahlen wie double Werte sind ebenfalls möglich (vgl. Oracle JAVA docs, Coordinates, Stand: 22.05.2016).

4.2.1. User Space

Zeichnet man ein Rechteck mit den Argumenten x = 0, y = 0, w = 500 und h = 500 wird dieses am Punkt (0,0) verankert. Die Breite und Höhe der Figur werden durch die Argumente w für die Breite und h für die Höhe definiert. Die Breite verläuft entlang der X-Achse, die Höhe entlang der Y-Achse. So ergeben sich für das Rechteck folgende Punkte: P1(0,0) P2(0,500), P3(500,500) und P4(500,0). In der nachfolgenden Abbildung 4.1. „Rechteck am Ursprung“ wird ein Rechteck durch die entsprechenden Punkte gezeichnet.

Das Device Space ist das Koordinatensystem des Ausgabemediums. Koordinaten referenzieren hier auf einzelne Pixel des Ausgabemediums, in diesem Fall der Bildschirm des Computers. Dieses Koordinatensystem kann sich stark von Gerät zu Gerät unterscheiden. Die Umrechnung vom User zum Device Space übernimmt die API automatisch während des Renderings[14] und muss nicht weiter definiert werden. Der Vollständigkeit halber sei an dieser Stelle das Image Space erwähnt. Dieses beschreibt das

[13] API zur Erstellung grafischer Oberflächen In Java
[14] Rendering beschreibt einen Prozess, bei dem aus einem Modell eine Grafik erzeugt wird.

lokale Koordinatensystem eines Bildes. Der zu verwendende Bereich eines Bildes wird durch die sogenannte Bounding Box definiert. Arbeitet der Programmierer mit Bildern, wird die Bounding Box vom Image zum User Space transformiert. Durch eigens definierte Transformationen können einzelne Bildausschnitte aus einem Bild in den User Space übertragen werden.

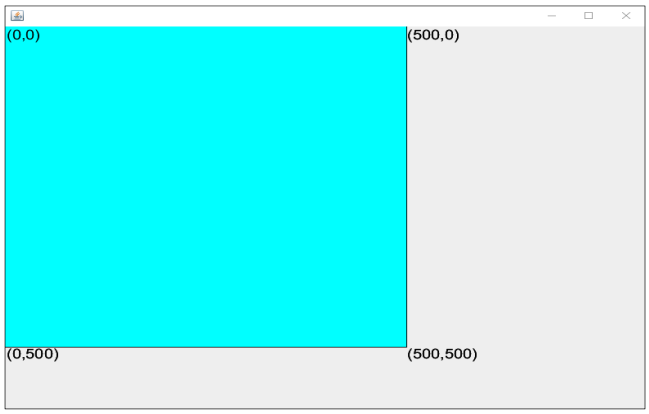

Abbildung 4.1 Rechteck am Ursprung

Die Koordinaten, die eine Form definieren sind unendlich dünn und liegen zwischen den Pixeln des Ausgabegerätes. Benutzt man eine Funktion, um den äußeren Rand einer Form zu zeichnen, beispielsweise *drawRectangle(x y w h)*, wird ein unendlich dünner Weg zwischen den Pixeln gezogen. Dieser Weg wird anschließend mit einem Stift in der definierten Pixelstärke nachgemalt. Der Stift setzt hierbei am Punkt an und überlappt die anliegenden Pixel in Richtung rechts unten. Somit muss bedacht werden, dass auf einer Zeichenfläche der Größe 800x600, wie in Abbildung 4.1. dargestellt, eine Linie mit dem Startpunkt (800,0) und Endpunkt (800,600) nicht zu sehen sein wird, da der Zeichenbereich rechts außerhalb des Bildes liegt (vgl. Internet: Oracle JAVA docs, Class Graphics, Stand: 22.05.2016).

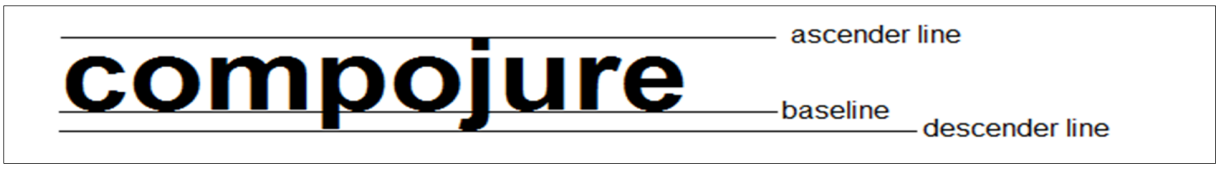

Abbildung 4.2 Schrift-Baseline (angelehnt an Oracle JAVA docs, 05.05.2016, Measuring Text)

Im Gegensatz zu Formen wird horizontaler Text ausschließlich über und nicht mit dem Ansatz unter der Baseline gezeichnet, wie es bei Schriftarten üblich ist. Die Punkt-Baseline beschreibt in diesem Fall nicht die normale Schrift-Baseline, sondern die Descender Line. Siehe hierzu Abbildung 4.2. Dies bedeutet, dass Buchstaben vollständig über der Punkt-Baseline gezeichnet werden (vgl. Internet: Oracle JAVA docs, Class Graphics, Oracle JAVA docs, Measuring Text, Stand: 22.05.2016). Daraus resultiert der in Listing 4.2 dargestellte Programmcode um obige Abbildung 4.1. „Rechteck am Ursprung“ mit Image-Compojure zu erstellen. Die Funktionen *rectangle* zeichnen hier zwei Rechtecke mit den oben beschriebenen Abmessungen auf die Zeichenfläche. Das erste Rechteck zeichnet den schwarzen Rahmen um das in cyan gefärbte Rechteck. *Styled-text* fügt die Punktbeschriftungen hinzu. Aus den oben genannten Gründen sind

die X- und Y-Koordinaten jeweils ein wenig nach unten bzw. rechts verschoben und entsprechen nicht den Koordinaten, die angezeigt werden.

```
(render (compose 800 600 {:text-antialiasing :on}
          (rectangle 0 0 500 500
             {:fill false :color (color :black) :width 1})
          (rectangle 1 1 499 499
             {:fill true :color (color :cyan)})
          (styled-text 11 50(create-styled-text "(0,0)"))
          (styled-text 501 50(create-styled-text "(500,0)"))
          (styled-text 501 517(create-styled-text "(500,500)"))
          (styled-text 11 517(create-styled-text "(0,500)"))))
```

Listing 4.2 Programmcode zu Abbildung 4.1

Für die Darstellung von Schriften kann zwischen logischen und physikalischen Schriftarten gewählt werden. Logische Schriftarten stehen auf allen Java Plattform zur Verfügung und besitzen unabhängig von der Plattform dasselbe Aussehen. Physikalische Schriftarten können zusätzlich hinzugefügt werden. Diese werden beispielsweise durch die Angabe eines Strings wie „Times New Roman“ bei der Instanziierung eines Font-Objekts erstellt. Bei der Verwendung von solchen Schriftarten besteht das Risiko, dass Texte auf einigen Plattformen nicht korrekt angezeigt werden können, da die Schriftart unter Umständen dort nicht zur Verfügung steht (vgl. Internet: Oracle JAVA docs, Physical and Logical Fonts, Stand: 22.05.2016).

4.2.2. Aufgaben

Zu den Aufgaben der 2D API gehört die Erstellung eines universellen Rendering Models[15], das auf unterschiedlichen Plattformen verwendbar ist. Auf Anwendungsebene ist der Rendering-Prozess plattformunabhängig und somit stets gleich. Hinzu kommt ein breites Spektrum an Möglichkeiten zum Erstellen von geometrischen Primitiven wie Rechtecken und Ellipsen oder Funktionen zum Erstellen selbstdefinierter Formen. Bei solchen Formen handelt es sich um Objekte der Klasse *java.awt.geom*, die als Shape-Objekte bezeichnet werden. Weiterhin kann definiert werden, wie sich gegenseitig überlappende Shapes dargestellt werden sollen - das sogenannte Compositing. Zusätzlich bietet die API Erleichterungen beim Farbmanagement, Unterstützung beim Drucken komplexer Dokumente und die Bestimmung diverser Rendering-Qualitätseinstellungen. Kernbestandteile der 2D API sind die beiden Java Klassen Graphics und Graphics2D. Diese werden im Folgenden näher betrachtet. Begonnen wird mit der Klasse Graphics (vgl. Internet: Oracle JAVA docs, Java 2D Rendering, Stand 22.05.2016).

[15] Rendering Model bezeichnet ein geometrisches Modell, aus dem eine Grafik erstellt werden kann.

4.2.3. Die Klassen Graphics und Graphics2D

Die abstrakte Klasse Graphics bildet die Basis für jede Klasse mit grafischen Kontext, die es Anwendungen ermöglicht auf Components wie zum Beispiel einem Canvas-Objekt[16] zu zeichnen.

Objekte der Instanz Graphics basieren auf dem State-Prinzip, das bereits im dritten Kapitel dieser Arbeit näher betrachte wurde. Der Zustand eines solchen Objekts wird durch die folgenden Eigenschaften angegeben (Namen des zuständigen Attributes in Klammern):

- ✓ Derzeitiges Component-Objekt auf das gezeichnet werden soll.
- ✓ Derzeitige Clipping Area, die den Bereich des Bildes, auf dem gezeichnet werden soll, definiert (Clip).
- ✓ Derzeitige Farbe, mit der gezeichnet wird (Paint).
- ✓ Derzeitige Schriftart, die für die Darstellung horizontalen Texts verwendet wird (Font).
- ✓ Derzeitige logische Pixeloperation (Composite)
- ✓ Derzeitige XOR Alternativ-Farbe (Composite)
- ✓ Derzeitige Rendering-Engine für Qualitätseinstellungen (Rendering Hints)

Werden Rendering-Operationen vorgenommen, sind nur Pixel von diesen betroffen, die sich im Bereich der derzeitigen Clipping Area befinden. Dieser Bereich wird durch eine Shape-Objekt definiert. Die Clipping Area des User Space wird zum Device Space transformiert, welche dann wiederum mit der Clipping Area des Device Space zusammengeführt wird. Diese Kombination definiert die finale Composite Clip, welche die Region des Bildes angibt, die gerendert wird (Vgl. Internet: Oracle JAVA docs, Class Graphics, Stand 22.05.2016).

Die Klasse Graphics2D (G2D) erweitert die Basisklasse Graphics um eine noch stärkere Kontrolle über geometrische Formen, Transformationen, Farbmanagement und Text Layout. Die zur Verfügung stehenden Rendering-Operationen sind Shape, Text und Image Operationen. Oracle beschreibt diese Klasse als:

> **"… fundamental class for rendering 2-dimensional shapes, text and images on the Java(tm) platform."** (vgl. Internet: Oracle JAVA docs, Class Grahphics2D, Stand 22.05.2016)

G2D ist in der Lage, mit den bereitgestellten Funktionen die Zustandseigenschaften, von nun an als Attribute bezeichnet, Clip, Paint, Font, Stroke, Transform, Composite und Rendering Hints zu manipulieren (vgl. Internet: Oracle JAVA docs, Class Graphics2D, Stand 22.05.2016).

[16] Ein leeres, rechteckiges Feld auf dem eine Anwendung gezeichnet werden oder ein Benutzer Ereignisse abfangen kann

Über das Attribute Stroke wird ein Linienstil definiert, mit dem anschließend eine Figur gezeichnet werden kann. Hierzu stehen die Breite des Stifts (width), das Aussehen der Linien-Endstücke von ungeschlossenen Figuren, die Teilstücke einer Linie (end caps), der Stil, mit denen Linien zusammengeführt werden (line joins), und die Option, Lücken in einer Linie einzubauen (dashes), zur Verfügung (vgl. Internet: Oracle JAVA docs, Stroking and Filling Graphics Primitives, Stand 22.05.2016).

Transform wird verwendet, um Shape-Objekte oder Bilder aus dem User Space zum Device Space zu transformieren. Das bedeutet, dass die derzeitige Transformation eines G2D-Objekts festlegt, an welcher Position das zu zeichnende Objekt im Device Space gerendert wird. Wendet man unterschiedliche Transformationen an, wird ein Objekt an anderen Positionen im Device Space gerendert. Das Beispiel in Abbildung 4.3 „Beispiel Translation“ zeigt ein Fenster, in dem zwei graue Rechtecke gerendert wurden. Wie in Listing 4.3 zu sehen besitzen im User Space beide Rechtecke, gezeichnet durch die Funktion *rectangle*, dieselben Koordinaten.

Jedoch wird für die Zeit des Aufrufs der ersten Funktion, die das rechte Rechteck zeichnet, die Transformation des G2D-Objekts mit einer neuen Transformation, in diesem Fall eine Translation entlang der X-Achse zum Punkt (400,0), konkateniert.

```
1 (render (compose 800 300 {:antialiasing :on}
2             (with-attributes {:color (color :light-grey)}
3                 (with-transform (transform
4                                   (translate 300 0))
5                     (rectangle 100 50 200 200))
6                 (rectangle 100 50 200 200))
7 (rectangle 100 50 200 200))
```

Listing 4.3 Programmcode zu Abbildung 4.3

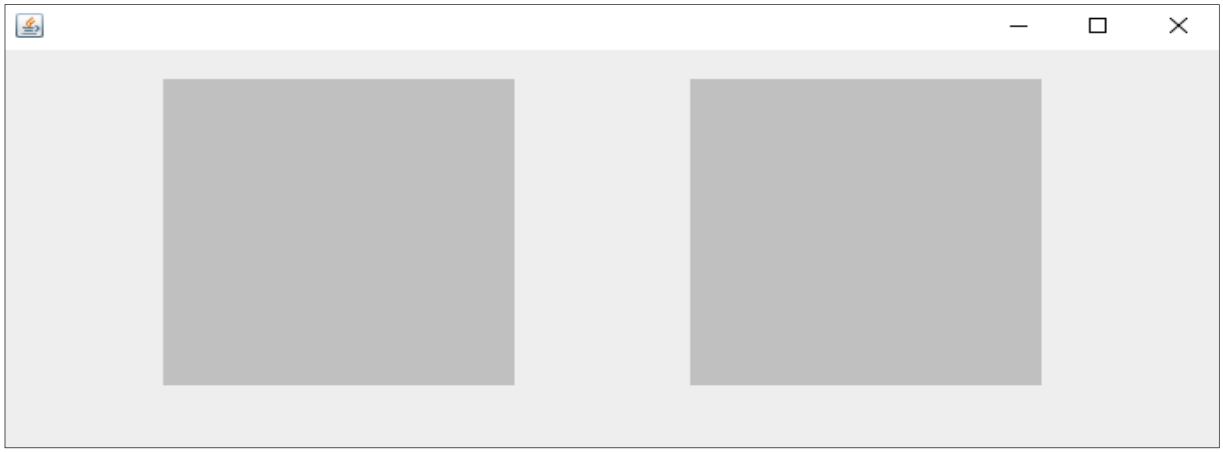

Abbildung 4.3 Beispiel Translation

Das Composite-Attribute beschreibt, welche Farbe im Zielpixel dargestellt werden wird. Man kann dadurch unterschiedliches Verhalten bei sich überlappenden Shapes erreichen. In Abbildung 4.4 „Beispiel Composite“ sind einige Attributwerte dargestellt. Der Quellcode zur Erstellung des Bildes kann im Materialverzeichnis A.1 eingesehen werden. Gerendert werden vier Rechtecke mit jeweils einem Kreis in der Mitte. Die Farbe des Kreises ist immer rot, die des Rechteckes schwarz. Durch Auswahl verschiedener Composite-Objekte als Attributwert wird bestimmt, von welchem Shape-Objekt die Farbe zum Einfärben des Pixels verwendet wird. Java zeichnet die Shapes entsprechend der Reihenfolge im Programmcode. Das letzte Shape-Objekt überzeichnet immer die vorherigen. Durch das Composite-Attribut kann die Zeichenreihenfolge manipuliert werden. Die Bilder A – D zeigen Ausgaben mit unterschiedlichen Composite-Objekten (vgl. Internet: Oracle JAVA docs, Compositing Graphics, Stand 22.05.2016).

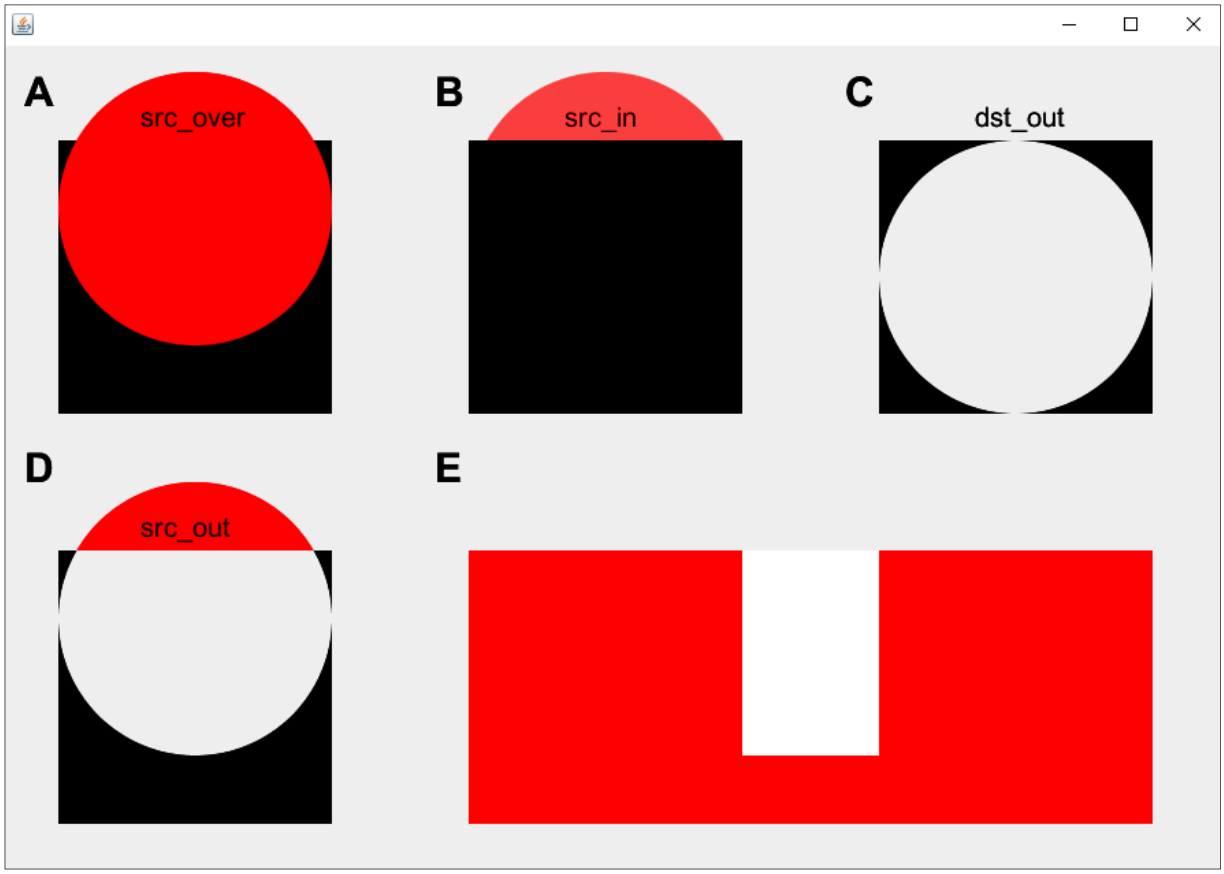

Abbildung 4.4 Beispiele Composite

Bei Bild E handelt es sich um eine spezielle Form des Composite Objekts. Diese beschreibt die Zustandseigenschaft XORMode und erstellt die Farbe der Zielpixel gemäß der Formel:

ZielPixel = (Pixel(QuellFarbe) ^ Pixel(xorFarbe) ^ ZielPixel);

Hierdurch ergibt sich für die Farbe der Pixel, die sich im Bereich des mittleren Rechtecks befinden, die Gleichung:

W(255, 255, 255) = R(255, 0, 0) ^ G(0, 255, 0) ^ B(0, 0, 255);

und somit die Farbe Weiß (vgl. Internet: Oracle JAVA docs, Class Graphics, Stand 22.05.2016).

Die letzte Eigenschaft Rendering-Qualität, wird durch Rendering Hints (Hinweise) definiert. Die Bezeichnung als Hinweise ist in der Eigenschaft begründet, dass diese die Rendering-Engine nicht fix definieren, sondern nur Hinweise auf die gewünschten Modi liefern. Kann die Rendering-Engine den Hinweisen nicht folgen, wird sie dies nicht. Eine mögliche Ursache hierfür sind Plattformen, die bestimmte Modi nicht unterstützen. Durch die Hinweise kann vorgegeben werden, ob Bilder so schnell wie möglich oder in der bestmöglichen Qualität gerendert werden sollen. Die folgenden Rendering Hints stehen in Java zur Verfügung und werden später in der DSL wählbar sein (vgl. JAVA 2d Graphics,1999 S.105-106):

1. **Antialiasing** : Verminderung eines unerwünschten Treppeneffekts / Kantenglättung für Formen.
2. **Alpha Interpolation** : Kontrolliert, wie Alpha Werte berechnet werden, wenn Shapes, Texte oder Bilder mit transparenten Farben zu rendern sind.
3. **Color Rendering** : Definiert, ob Farbkorrekturen für ein vorhandenes Display vorgenommen werden sollen.
4. **Dithering** : Nimmt die Berechnung approximierter Farben vor, sofern das Ausgabemedium nicht alle benötigten Farben rendern kann.
5. **Fractional Text Metrics** : Ermöglicht die Wahl zwischen Floating Point oder Integer Werten für Schriftgrößen.
6. **Rendering** : Ermöglicht, die Wahl den Rendering-Prozess so schnell oder so gut wie möglich durchzuführen.
7. **Interpolation** : Definiert den Algorithmus, der für die Transformation von Bildern verwendet wird.
8. **Stroke Normalization Control** : Kontrolliert, ob die Geometrie von gerenderten Shapes zu verschiedenen Zwecken angepasst werden dürfen.
9. **Text Antialiasing** : Antialiasing nur für Text, da dies zu starken Einbußen der Rechenleistung führen kann.

5. DSL Image-Compojure

In diesem Teil der Arbeit wird die eigens programmierte DSL Image-Compojure eingehend analysiert. Zunächst erfolgt die Definition der Anforderungen, die an die neue Sprache gestellt werden. Darauf folgt die Beschreibung der erdachten Grammatik sowie dessen Verwendung. Ergänzend hierzu wird das Design der DSL und dessen Implementierung näher betrachtet. Den Schluss bildet ein Ausblick auf die geplante Weiterentwicklung sowie auf den ersten realen Anwendungsfall anhand eines kleinen Beispiels.

5.1. Anforderungen

Die spezifische Domäne der DSL Image-Compojure wird durch das Erstellen eines Bildes unter Verwendung der Java 2D API abgebildet. Dabei werden einige Anforderungen an die DSL gestellt. Diese Anforderungen sind in zwei Bereiche aufteilt. Der eine ist an die Funktionalität der Sprache geknüpft. Dieser beschreibt, welche Möglichkeiten einem Entwickler zur Verfügung stehen, um Bilder zu erstellen. Während sich der erste Teil mit dem Gegenstand (Was) beschäftig, widmet sich der zweite Part dem Verhalten (Wie) der Sprache. Das Verhalten definiert die Anforderungen an das Design der Sprache.

5.1.1. Gegenstand (Was?)

Dem Programmierer muss es möglich sein,

- ✓ ein Bild in einer gewünschten Größe zu erstellen.
- ✓ Shape-Objekte mit definierten Farben und Stroke-Attributen zu zeichnen.
- ✓ Farben, Texturen und Farbverläufe zu definieren.
- ✓ Standardisierte Attributwerte zur Manipulation des Zustands des G2D-Objekts anzugeben, die auf alle nachfolgenden Shapes, Schriftzüge und Bilder angewendet werden.
- ✓ die Qualität des Ausgabebildes, durch Anpassen der Rendering Engine zu bestimmen.
- ✓ das Farbverhalten von sich überlappenden Figuren anzugeben (Compositing).
- ✓ Schriftzüge durch verschiedene Schriftarten, Styles, Farben und Größen zu erstellen.
- ✓ Schriftzüge zu zeichnen.
- ✓ Transformationen für Verschiebung, Drehung, Scherung uns Skalierung zu erstellen.
- ✓ Transformationen auf alle nachfolgenden Shapes, Schriftzüge und Bilder anzuwenden.
- ✓ Transformationen in einander zu verschachteln.
- ✓ Bilder zu skalieren.
- ✓ Bilder im Dateisystem zu speichern.

- ✓ Bilder aus dem Dateisystem zu laden.
- ✓ Bilder in einem Fenster anzuzeigen.
- ✓ viele Bilder gleichzeitig zu rendern unter Berücksichtig von Multithreading.
- ✓ die Clipping Area anzugeben.

5.1.2. Verhalten (Wie?)

Das Design der DSL muss die folgenden vier Kriterien erfüllen:

1. Schnell zu erlernen und einfach anzuwenden
2. Clojure Affinität
3. G2D-Zustand verschleiern
4. Threadsicherheit

Das erste Kriterium zum schnellen Erlernen und einfachen Anwenden ist bereits erfüllt, da eine DSL, wie bereits festgestellt wurde, auf Grund der geringen Komplexität schneller zu erlernen ist als eine GPL. Dennoch kommt es häufig vor, dass eine Sprache um Funktionen erweitert wird, die nicht domänenspezifisch sind. Die Sprache wird somit unnötig komplex und schwerer zu erlernen. Die Syntax der Sprache muss einheitlich designend sein. So sollten Namen von Funktionen stets derselben Konvention folgen und nicht beispielweise bei Worttrennungen zunächst ein (–) und andernorts ein (_) verwenden. Zusätzlich gilt ist es, die Verwendung von abwechselnder Groß- und Kleinschreibung zu vermeiden. Ebenfalls sollte sich die Sprache bei der Übergabe von Argumenten an Funktionen an einer einheitlichen Reihenfolge orientieren. Ein Gegenbeispiel hierzu zeigt, was einem Entwickler das Lernen erschweren könnte:

- ✓ *(rectangle [x y w h] (.drawRectangle g2d x y w h))*
- ✓ *(round-rectangle [w h x y] (.drawRoundRectangle g2d x y w h))*

Hier wurden die Koordinaten, die den obersten linken Punkt des Rechtecks definieren, mit den Werten, die die Höhe und Breite angeben, vertauscht. Dies führt bei Angabe von gleichen Werten zu ungewollten Ergebnissen. Das Argument *g2d* kann in diesem Zusammenhang ignoriert werden. Erleichtert wird das Erlernen und Anwenden zusätzlich durch eine gute Dokumentation der Sprache. Clojure bietet hier durch die Docstrings eine sehr komfortable Form der Dokumentation. Bestenfalls gestaltet sich der Quellcode zum Erhalt von Ergebnissen um ein Vielfaches kürzer, als ursprünglich bei der Verwendung einer GPL nötig gewesen wäre. So sollte es im vorliegendem Fall nicht mehr als eine Codezeile benötigen, um ein Bild einer festen Größe in einer neuen Größe zu erhalten.

Im Kapitel 4.1 „Domain Specific Language“ wurde erwähnt, dass während eine API lediglich Worte und Satzzeichen liefert, eine DSL die Grammatik hinzufügt. Das bedeutet, dass sich die Makros und Funktionen verschachteln lassen müssen. Was jedoch nicht dazu führen darf, dass der Quellcode komplizierter und schwerer zu lesen sein ist. Die Anwendung muss trotz allem leichtfallen. Wie genau das erreicht wird, ist nachfolgendem Kapitel 5.2 „DSL Grammatik und Verwendung“ näher erläutert.

Mit dem zweiten Kriterium der Clojure Affinität geht einher, dass der Entwickler, obwohl er sich im Kontext der DSL befindet, weiterhin normalen und validen Clojure-Code verwenden kann. Die DSL darf den Entwickler nicht in der Verwendung von Funktionalitäten beschränken. Genauso wichtig ist es, dass die Sprache ähnlich zu dem nativen Clojure-Code erscheint. Es wird folglich keine neue Syntax eingeführt, sondern die Bestehende erweitert. Zusätzlich werden alle Java Interops in Wrapper-Funktionen[17] gekapselt, sodass auch diese wie normale Clojure Funktionen aussehen.

Die Wrapper-Funktionen führen direkt zum dritten Kriterium, die Verschleierung des G2D-Zustands. Wie beschrieben basiert die 2D API zum Großteil darauf, den Zustand des Objekts zu verändern und anschließend Rendering-Operationen vorzunehmen. Da dies jedoch nicht dem Grundsatz der funktionalen Programmierung entspricht, wird versucht, den Zustand des Objekts vor dem Entwickler zu verbergen (vgl. Kapitel 3.1). Manipulationen am Objekt erfolgen automatisch im Hintergrund und werden nach Abschluss einer Operation wieder rückgängig gemacht. Der Entwickler muss sich somit nicht weiter darum kümmern, ob sich das Objekt im korrekten Zustand befindet, um gewünschte Ergebnisse zu liefern.

Das vierte Kriterium der Threadsicherheit soll gewährleisten, dass es der DSL möglich ist, Bilder parallel in verschiedenen Threads zu rendern, ohne dass sich diese Threads dabei gegenseitig stören. Probleme, die üblicher Weise bei Anwendungen mit multiplen Threads auftreten, wie sie im Kapitel 3.1.2 beschrieben sind, sollen vermieden werden.

[17] Interop-Funktionen werden in normalen Clojure Funktionen gekapselt.

5.2. Grammatik und Verwendung

In diesem Abschnitt wird nun auf die eigens erdachte Grammatik eingegangen und gleichzeitig dargelegt, wie die DSL Image-Compojure richtig verwendet wird. Erläuterungen werden anhand von Quellcode-Beispielen vorgenommen. Um das Verständnis zu erleichtern, liegen dem Material dieser Arbeit Rendering-Ausgaben zu den Beispielen bei. Hierbei handelt es sich um Screenshots der gerenderten Bilder, die in einem JFrame angezeigt werden.

Bei der Beschreibung wird der Begriff Kontext in Verbindung mit Makros, sowie der Begriff Rendering-Operation häufiger Verwendung finden. Der Kontext eines Makros beschreibt nach eigner Definition alle Funktionsaufrufe, die sich zwischen den Klammern des Makros befinden *(Makro-Name Args* (Funktionsaufrufe))*. Als Rendering-Operation werden alle Funktionen bezeichnet, die Shape-Objekte, Bilder oder Texte auf der Zeichenfläche darstellen. Schließlich findet der Begriff Key-Value-Paare Verwendung. Um Unklarheiten zu vermeiden, sei an dieser Stelle erwähnt, dass Values von Keys in einer Map auch durch Keys beschrieben werden können. Im Abschnitt 5.3 „Design und Implementierung“ wird dies näher erläutert.

5.2.1. Compose

Das Kernstück der DSL bildet das Makro *compose*. Alle Funktionen, die Rendering-Operationen vornehmen, tun dies in dem Grafikkontext, der von diesem Makro definierten wird. Die vorliegende DSL kann auch ohne das *compose* Makro verwendet werden. Hierzu existiert im Hintergrund ein Standardbild, das einen Standard-Grafikkontext zur Verfügung stellt. Wird jedoch das *compose* Makro nicht benutzt, kann es bei der Verwendung von multiplen Threads zu ungewünschten Ergebnissen kommen. Wie dieses Verhalten zu erklären ist, wird im Abschnitt 5.3 „Design und Implementierung“ im Detail beschrieben. Beim Aufruf von *compose* muss zunächst die Größe des neuen Bildes bestimmt werden. Aus dem Bild wird anschließend der Grafikkontext erstellt. Das erste Argument bestimmt die Breite, das zweite die Höhe des Bildes, das dritte Argument ist optional. Wird das dritte Argument (von nun an als Render-Settings bezeichnet) angegeben, dann gibt es die Rendering Hints an. Diese Render-Settings müssen als Map mit Key-Value-Paaren übergeben werden. Unbekannte Keys und Values werden ignoriert. Alle verfügbaren Keys mit dazugehörigen Values sind in Tabelle 5.1 (S. 32) „Render-Settings“ aufgeführt (vgl. Kapitel 4.1.3). Den Optionen folgt eine beliebige Anzahl an Funktionen. Schließlich gibt das Makro ein Bild vom Typ *java.awt.image.BufferedImage* zurück. Listing 5.1 (S. 32) zeigt wie das *compose* Makro im Programmcode verwendet wird.

```
(compose
  800 600
  {:antialiasing          :off
   :aplpha-interpolation :default
   :color-rendering      :quality
   :dithering            :disable
   :fractional-metrics   :on
   :interpolatioin       :bicubic
   :rendering            :default
   :stroke-control       :default
   :text-antialiasing    :default
   }
      (functions-body))
```

Listing 5.1 Anwendung compose Makro

Attribut	Key	Value	Attribut	Key	Value
Antialiasing	:antialiasing	:on :off :default	Stroke-Control	:stroke-control	:normalize :default :pure
Alpha-Interpolation	:alpha-interpolation	:quality :speed :default	Rendering	:rendering	:quality :speed :default
Color-Rendering	:color-rendering	:quality :speed :default	Text-Antialiasing	:text-antialiasing	:on :off :default :gasp :lcd-hrgb :hbgr :lcd-vrgb :lcd-vgbr
Dithering	:dithering	:disable :enable :default			
Fractional-Metrics	:fractional-metrics	:on :off :default			
Interpolation	:interpolation	:bicubic :bilinear :neighbor			

Tabelle 5.1 Render-Settings

5.2.2. Shapes

Funktionen zum Zeichnen von Shape-Objekten, folgen immer demselben Aufbau. Zunächst werden mit den ersten Argumenten die Koordinaten des zu zeichnenden Objekts angegeben. Die Anzahl der Argumente für Koordinaten variieren an dieser Stelle, abhängig von der Art des Shapes, von Funktion zu Funktion. Auf die Koordinaten folgt eine Map, die mittels Key-Value-Paaren Attribute definiert. Diese Map ist im Programm als *attributes* benannt, weshalb von nun an der Begriff Shape-Attribute verwendet wird. Shape-Attribute spezifizieren Attribute, die das Aussehen von Shapes, wie beispielweise die Linienart oder Farbe, auf der Zeichenfläche vorgeben. Zum Zeichnen von vordefinierten Shapes stehen die Funktionen:

✓	*(line*	*x1*	*x2*	*x2*	*y2*			*attributes*	*)*
✓	*(rectangle*	*x1*	*y2*	*w*	*h*			*attributes*	*)*
✓	*(round-rectangle*	*x1*	*y2*	*w*	*h*	*arcW*	*arcY*	*attributes*	*)*
✓	*(oval*	*x1*	*y2*	*w*	*h*			*attributes*	*)*
✓	*(polyline*	*seqX*	*seqY*	*numCo*				*attributes*	*)*
✓	*(polygone*	*seqX*	*seqY*	*numCo*				*attributes*	*)*

zur Verfügung. Zusätzlich ist es möglich, mittels der Funktion *(shapes ShapeSeq attributes)* mehrere vordefinierte Shape-Objekte zu zeichnen. Diese Objekte werden in Form eines Vektors oder einer Liste der Funktion als erstes Argument übergeben. Allen Funktionen kann das optionale Argument *attributes* zur Spezifikation der Shape-Attribute mitgegeben werden. Im Argument *attributes* steht der Key *:fill* zur Verfügung. Ist der zugehörige Wert *true* oder vorhanden und somit ebenfalls *true*, wird das Shape-Objekt ausgefüllt (vgl. Kapitel 3.3). Wird dieser Key nicht angegeben, ist der Wert *nil*, wodurch lediglich die Umrandung des Shapes gezeichnet wird. Listing 5.2 zeigt die Verwendung der Funktion *shapes*. In diesem Beispiel wird der Funktion ein Vektor mit zwei Shape-Objekten, einem Rechteck und einem Polygon, übergeben. Diese werden mit der Farbe Gelb gefüllt, da der Key *:fill* mit dem Wert *true* und *:paint* mit dem Wert *:yellow* in den Shape-Attributen definiert ist.

```
1(shapes [(Rectangle2D$Double. 0 0 100 100)
2         (Polygon. (int-array [150 250 325 375 450 275 100])
3                   (int-array [150 100 125 225 250 375 300]) 7)]
4                       {:fill true :paint :yellow}))
```

Listing 5.2 Beispiel shape Funktion

5.2.3. Farben, Verläufe, Texturen

Für den Wert des Paint-Attributs, das mit dem Key *:paint* gesetzt wird, stehen drei verschiedenen Objekte zu Verfügung, die mit jeweils eigenständigen Methoden erstellt werden. Dazu gehören:

- ✓ **Farbe** (*color key*
r g b 1
r g b a)
- ✓ **Farbverlauf** *(gradient-paint x1 y1 colorA x2 y2 colorB cyclic*
x1 y1 colorA x2 y2 colorB)
- ✓ **Textur** *(texture txtr anchor-x1 anchor-y1 anchor-x2 anchor-y2)*

Neben einheitlichen Farben ist es möglich Texturen und Farbverläufe zu verwenden. Zum Erstellen von einfachen Farben stehen dem Anwender vier verschiedene Möglichkeiten zur Verfügung. Der Aufruf von *color* ohne Argumente, wird das Objekt *java.awt.Color.WHITE* zurückliefern. Analog dazu, kann durch die Angabe eines der verfügbaren Keys *:green, :blue, :red, :black, :yellow, :pink, :orange, :magenta, :light-grey, :dark-grey* oder *:cyan* ein Java Color-Objekt der jeweiligen Farbe erstellt werden. Reicht die Anzahl der Standard-Farben nicht aus, können eigene Color-Objekte durch die Angabe von RGBA Werten im Bereich 1 – 255 erstellt werden. Wird der Alpha Kanal weggelassen, ist dieser Wert standardmäßig 255 und somit 100% opak, bzw. nicht durchlässig.

Zum Erstellen eines Farbverlaufs müssen der Funktion zwei Punkte und zwei Farben übergeben werden. Der Farbverlauf verläuft somit vom ersten Punkt mit Farbe A zum zweiten Punkt zur Farbe B. Wird das Argument *cyclic* mit Wert *true* übergeben, wiederholt sich der Farbverlauf vor Punkt 1 und nach Punkt 2 zyklisch. Ist der Wert *false* oder *nil*, erscheinen die Pixel vor und nach den Punkten konstant, entsprechend der übergebenen Farben.

Als Letztes folgt die Verwendung einer Textur, weshalb zunächst ein BufferedImage geladen werden muss. Hier bietet sich die *load-image* Funktion an, auf die im weiteren Verlauf noch eingegangen wird. Der *texture* Funktion muss dieses Bild, sowie die Koordinaten, die eine Fläche spezifizieren, auf welche das Bild projiziert wird, übergeben werden. Listing 5.3. zeigt, wie Farben, Texturen und Farbverläufe erstellt und anschließend zum Zeichnen verwendet werden können. Material A.2 zeigt das entstehende Bild.

```
1 (defn color-example [shapes1 shapes2 shapes3]
2   (let
3   [colors [(color :black) (color 255 0 0 125) (color 0 0 255)]
4    texture (texture-paint (load-image "res/txt.png") 0 0 50 50)
5    gradient (gradient-paint 200 200(first colors)600 600(second colors))]
6       (shapes shapes1 {:paint (first colors)})
7       (shapes shapes2 {:paint texture})
8       (shapes shapes3 {:paint gradientpaint})))
```

Listing 5.3 Erstellung von Paint Values

Hier werden drei Shape-Objekte mit einer einfachen Farbe, einer Textur und einem Farbverlauf gezeichnet. Die Shape-Objekte werden der hier definierten Funktion *color-example* übergeben, und die Werte für die Paint-Attribute werden in einem Let-Block gebunden.

5.2.4. With-attributes

Die Angabe der Shape-Attribute ist optional, da die notwendigen Attribute Standardwerte besitzen. Diese Standardwerte können mit Hilfe des *with-attributes* Makros überschrieben werden. Alle Rendering-Operationen, die im Kontext dieses Makros stattfinden, werden mit den spezifizierten Shape-Attributen ausgeführt. In Listing 5.4 ist die Verwendung des Makros exemplarische dargestellt.

```
1 (with-attributes
2   {:width       1.0
3    :join        :miter
4    :miter-limit 10.0
5    :cap         :square
6    :dash        nil
7    :dash-phase  0
8    :composite   :src_over
9    :alpha       1.0
10   :paint       (color 255 0 0 255)
11   :xor-mode    nil
12   }
13     (functions-body))
```

Listing 5.4 Anwendung with-attributes Makro

Hier sind alle zur Verfügung stehenden Keys angegeben. Wird das Makro verwendet, ist es nicht zwingend notwendig alle Keys zu spezifizieren. In diesem Fall werden die Standardwerte der Attribute von nicht angegeben Keys, mit den neuen Attributwerten verschmolzen und entsprechend gesetzt. Bei diesen handelt es sich um die gleichen Keys und Values, die auch in den zuvor beschriebenen Funktionen für Rendering-Operationen abzüglich des Keys *:fill* verwendet werden können. Alle verfügbaren Keys mit zugehörigen Values finden sich in der Tabelle 5.2 „Shape-Attribute“ (S. 36). Unbekannte Key-Value-Paare werden wie auch beim *compose* Makro ignoriert.

Verwendet der Entwickler das *with-attributes* Makro und gibt einer Funktion zusätzlich noch Spezifizierungen der Attribute mit, überschreiben die der Funktion mitgegebenen Werte, die Werte des Makros. Nach Abschluss der Rendering-Operation werden wieder die Werte des Makros für alle nachfolgenden Operationen verwendet.

Attribut	Key	Value	Attribut	Key	Value
Stroke-caps	:cap	:butt :round :square	Stroke-joins	:join	:bevel :round :miter
Composite	:composite	:src :src_in :src_out :src_over :dst_in :dst_out :dst_over :clear	Width	:width	float
			Miter-limit	:miter-limit	float
			Dash	:dash	vector / list
			Dash phase	:dash-phase	float
			Alpha	:alpha	float
			Paint	:paint	Color-Key, Color, TexturePaint, GradientPaint

Tabelle 5.2 Shape-Attribute

```
1 (defn with-attributes-example []
2   (with-attributes {:paint :yellow :width 50}
3             (oval 50 50 500 500 {:paint :black :fill true :width 200})
4             (oval 50 50 500 500 )))
```

Listing 5.5 Beispiel simpler Rendering-Operationen

Der Quellcode in Listing 5.5 erzeugt durch die Funktion *oval* zwei Kreise. Zunächst wird eine schwarze Kreisfläche gezeichnet. Anschließend wird das Bild um einen gelben Ring mit einer Linienstärke von 50 Pixeln erweitert, der die schwarze Kreisfläche umschließt. Die gelbe Farbe und die Linienstärke erhält der Ring durch das *with-attributes* Makro, welches die Funktionen einschließt. Durch die Angabe der Keys *:paint* und *:fill* mit den Values *:black* und *true* als Attributwert beim Aufruf der ersten *oval* Funktion, werden die vorher festgelegten Attribute für diese Operation überschrieben und der Kreis schwarz ausgefüllt. Nachdem die Kreisfläche gezeichnet wurde, werden die Attributwerte automatisch wieder zurückgesetzt. Anzumerken ist hier, dass der Key *:width* mit Value *200* keine Auswirkungen hat, da keine Linie bei diesem Aufruf gezeichnet wird (für Beispiel siehe Material A.3).

Das Makro *with-attributes* kann verschachtelt verwendet werden. Das bedeutet, dass im Kontext des Makros dieses beliebig oft aufgerufen werden kann. Eine Beschränkung der Tiefe ist nicht gegeben. Auf diese Weise muss der Entwickler nicht erneut alle Shape-Attribute spezifizieren, da beispielsweise alle nachfolgenden Shapes eine neue Farbe, die übrigen Attribute jedoch weiterhin dieselben Werten besitzen sollen.

5.2.5. With-transform

Wie auch beim *with-attributes* Makro ist eine Verschachtelung bei der Verwendung des *with-transform* Makros möglich. Doch zunächst wird das Makro *transform* näher erläutert. Dieses erzeugt ein AffineTransform-Objekt und konkateniert alle nachfolgenden Transformationen miteinander. Zur Verfügung stehen die Transformationen:

- ✓ Verschiebung *(translate tx ty)* Transformationsmatrix: $\begin{vmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{vmatrix}$
- ✓ Scherung *(shear shx shy)* Transformationsmatrix: $\begin{vmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$
- ✓ Skalierung *(scale sx sy)* Transformationsmatrix: $\begin{vmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{vmatrix}$
- ✓ Rotation *(rotate theta, vecx vecy, theta anchorx, anchory, vecx vecy anchorx anchory)*

  Transformationsmatrix: $\begin{vmatrix} \cos(theta) & -\sin(theta) & 0 \\ \sin(theta) & \cos(theta) & 0 \\ 0 & 0 & 1 \end{vmatrix}$

Die Argumente der Verschiebung, Scherung und Skalierung definieren die Faktoren, mit denen sich die neuen Koordinaten entlang der X- und Y-Achse berechnen. Bei der Rotation können entweder der Drehwinkel oder ein Drehvektor übergeben werden. Zusätzlich ist es möglich, einen Punkt zu bestimmen, um den eine Shape-Objekt gedreht werden soll. In diesem Zusammenhang wird das Shape-Objekt erst an den gewünschten Punkt verschoben, anschließend gedreht und schließlich wieder zurückgeschoben. Wurden alle Transformationen miteinander konkateniert, wird das AffineTransform-Objekt zurückgegeben.

```
1(with-transform (transform
2                        (translate 100 100)
3                        (rotate 10)
4                        (shear 100 100)
5                        (scale 100 100)
6       (functions-body))
```

Listing 5.6 Transformationsbespiel

Das entstandene Objekt kann nun an *with-transform* übergeben werden. Listing 5.6 zeigt, wie das im Programmcode aussehen kann. In diesem Beispiel wird eine Transformation erstellt, die einen Drehung,

Rotation, Scherung und Skalierung beinhaltet. *With-transform* konkateniert die derzeitige Transformation[18] mit der des neuen AffineTransform-Objekts. Alle nachfolgenden Shapes werden nun mit dieser neu entstandenen Transformation im Device Space gerendert.

Aus diesem Grund kann auch *with-transform* unendlich tief verschachteltet werden. Transformationen werden niemals überschrieben, sondern konkateniert. Ist der Kontext des Makros beendet, wird wieder die vorherige Transformation verwendet. Wie dies funktioniert, wird im Kapitel 5.4 erläutert.

5.2.6. Bilder

Transformationen können ebenfalls auf Bilder angewendet werden. Zum Zeichnen von Bildern stehen die folgenden Funktionen zur Verfügung:

- ✓ *(image x y image,*

  dstx1 dsty1 dstx2 dsty2 srcx1 sry1 srcx2 srcy2 image) und
- ✓ *(load-image src)*

```
1(defn image-example []
2  (let [img (load-image "res/test.png")]
3    (image  0 0 img )
4    (image  0 0 (/(.getWidth image) 2) (/(.getHeight image) 2)
5            0 0 (.getWidth image)(.getHeight image) image )))
```

Listing 5.7 Verwendung image

Die Funktion *load-image* lädt ein Bild als BufferedImage aus dem Dateisystem. Das Argument *src* gibt den Pfad zum Bild an. Die Funktion *image* zeichnet ein geladenes Bild auf die Zeichenfläche des aktuellen Grafikkontexts. Hierbei kann das Bild in der Originalgröße direkt an einem Punkt, spezifiziert durch die Argumente *x* und *y*, oder auch skaliert und mit einem bestimmten Bildausschnitt, gezeichnet werden. Die Skalierung erfolgt zum einen durch die Angabe einer Zielfläche mit einem Ansatzpunkt, sowie durch die Größe des Rechteckes (*dstx1 dsty1 dstx2 dsty2*) im User Space des zu aktuellen Zeichenkontexts. Zum anderen durch die Definition der Bounding Box im Image Space des zu zeichnenden Bildes, welches durch die Argumente *srcx1 sry1 srcx2 srcy2* angegeben wird. Ist die Fläche der beiden Rechtecke äquivalent, wird keine Skalierung vorgenommen. Ist die Zielfläche größer, wird das Bild vergrößert. Sollte diese kleiner sein, wird das Quellbild verkleinert. Das Listing 5.7 zeigt eine Funktion, die zunächst in einem Let-Block über die *load-image* Funktion das Bild *res/test.png* an das Symbol *img* bindet und anschließend das Bild

[18] Der Grafikkontext des Bildes besitzt eine Standard-Transformation, mit der die Konkatenation vorgenommen wird.

über die *image* Funktion an den Punkt (0,0) im User Space zeichnet. Mit dem zweiten Aufruf wird das Bild an derselben Stelle gezeichnet, jedoch um 50% verkleinert.

Es ist zusätzlich möglich, Anpassungen der Größe eines Bildes mit der Funktion *(resize image-to-scale w h)* vorzunehmen. Die Funktion liefert ein BufferedImage der Größe *w* für die Breite und *h* für die Höhe des hineingegebenen Bildes *img* zurück. Dies ist eine Vereinfachung zur Verwendung der komplexeren *image* Funktion, sofern ein komplettes Bild skaliert werden soll.

5.2.7. Schriftzüge

Neben den Funktionen zur Verwendung von Shapes, Transformationen und Bildern bietet Image-Compojure auch die Möglichkeit, Schriftzüge auf einem Bild darzustellen. Zu diesem Zweck existieren zwei Funktionen. Mit der Funktion *create-styled-text* kann ein Schriftzug erstellt und dessen Erscheinungsbild angepasst werden. In Listing 5.8 ist dargestellt, wie beide Funktionen zusammen Verwendung finden. Das Listing zeigt die Funktion *create-styled-text* mit allen verfügbaren Argumenten. Aufrufe sind wie folgt möglich:

(create-styled-text string
string font style
string font style size)

```
1  (styled-text 50 200 (create-styled-text
2                       "IMAGE-COMPOJURE"
3                       :times :bold 50
4                      {:foreground (color :yellow)
5                       :background (color :red)
6                       :kerning false
7                       :strikethrough false
8                       :swap-colors false
9                       :underline :low-on-pixel
10                      :weight :weight-demibold
11                      :width :width-condensed})))
```

Listing 5.8 Verwendung create-styled-text

Sind Argumente nicht angegeben, werden automatisch Standardwerte übergeben. Durch die Verwendung des letztens Arguments kann der Stil des Schriftzuges noch feiner spezifiziert werden. Dies funktioniert analog zu den bisher beschriebenen Attributen, jedoch finden hier andere Keys und Values Verwendung. Unbekannte Keys und Values werden an dieser Stelle ignoriert. In der Tabelle A.1 „Attribute für Schrift“ (Material A.4) sind alle Attribute aufgelistet, die das Erscheinungsbild eines Schriftzuges beschreiben. Ist in der Spalte Arg/ Key ein Key angegeben, muss dieser in der Map (ab Z. 4) verwendet werden. Andernfalls handelt es sich um ein Argument (Z.3), dessen Value direkt beim Aufruf der Funktion *create-styled-text* an

der entsprechenden Stelle übergeben werden muss. Als Ergebnis des Aufrufs *create-styled-text* in Listing 5.8 Z.3 (Ausgabe in Material A.5) erhält der Entwickler eine Map, die an (*styled-text x y text-map)* übergeben wird. Beim Aufruf der Funktion *styled-text* können keine Attribute angegeben werden. *Styled-text* verwendet die zuvor definierte Schriftart und bildet den Schriftzug auf der Zeichenfläche des aktuellen Grafikkontexts ab. Die Argumente *x* und *y* bestimmen den Punkt, in diesem Fall (50,200), von dem aus der Schriftzug gezeichnet wird. Schriftzüge verhalten sich anders als bisher beschriebene Rendering-Operationen. Das *with-attributes* Makro hat nur Auswirkungen auf das Attribut Composite. Fest definierte Stile können leicht wiederverwendet werden. So ist es möglich, in Verbindung mit der Funktion *merge* andere Schriftzüge mit dem gleichen Schriftstil zu verwenden.

5.2.8. Ausgabe

```
1(defn render-example [image]
2  (render image {:as :file :path "res/output.png" :clipping shape})
3  (render image {:as :b64 })
4  (render image))
```

Listing 5.9 Verwendung render-output

Zur Ausgabe des Bildes, das *compose* zurückliefert, dient die Funktion *render*. Dieser Funktion werden ein Bild sowie die gewünschte Art der Ausgabe übergeben. In Listing 5.9 sind die verschiedenen Arten zur Ausgabe dargestellt. Die Art der Ausgabe wird durch eine Map über den Key :as angegeben. Zur Auswahl stehen die Values *:file, :b64* und *:show*. Durch die Angabe des Values *:file* wird das Bild im Dateisystem gespeichert. Hierfür wird der zusätzliche Key *:path* benötigt, der den Pfad angibt, in dem das Bild gespeichert wird. Bei Erfolg gibt die Funktion *nil*, bei Misserfolg die aufgetretene IOException zurück. Der Value *:b64* hingegen sorgt dafür, dass das BufferedImage in einen String im Base64 Format codiert wird. In diesem Fall wird der String zurückgeliefert. Als letzte Möglichkeit steht der Key :show zur Verfügung. Ist keine Map beim Aufruf angegeben, wird automatisch die Logik ausgeführt, die sich hinter *:show* verbirgt. Diese zeigt das Bild in einem JFrame an und gibt den JFrame zurück. Anschließend kann dieser JFrame an die Funktion *(repaint JFrame)* übergeben werden, welche das Bild im Frame neu zeichnet. Auf diese Weise können Animationen erstellt werden. Diese sind jedoch im Rahmen der Arbeit und in den Anforderungen nicht vorgesehen, weshalb auf Animationen nicht weiter eingegangen wird.

5.2.9. Uneingeschränkte Clojure Funktionalität

Das in Listing 5.10 aufgeführte Beispiel dient zur Verdeutlichung, dass ein Entwickler im Kontext des *compose* oder *with-attributes* Makros, uneingeschränkt auf die Funktionalitäten und Syntax der Sprache Clojure zurückgreifen kann. Zu diesem Zweck wird hier eine Funktion *example* erstellt.

In der Funktion erfolgt zunächst der Aufruf von *render*, der ein Bild übergeben wird, welches von compose erstellt wird. In diesem Bild werden Linien gezeichnet. Alle Linien, deren Faktor *n* zur Berechnung der zweiten X-Koordinate gerade ist, werden rot, die anderen blau gezeichnet. Die rote Farbe wird durch *with-attributes* gesetzt, die blaue durch die Angabe des Keys *:paint* in den Shape-Attributen des zweiten Funktionsaufrufs von *line* in Zeile 9. Über die If-Bedingung wird definiert, welcher Aufruf von *line* gewählt wird. Dieser Block wird wiederum als anonyme Funktion der Funktion *map* übergeben. *Map* ruft nun jeweils die anonyme Funktion für den Wertebereich 100-106 auf. Da *map* nur eine lazy sequence[19] zurückliefert, wird durch die Funktion *doall* die tatsächliche Evaluation erzwungen. *Dotimes* ruft wiederum *map* zehn Mal mit dem Wertebereich 0-10 auf. Die Werte *n* und *val* aus den Wertebereichen werden dann in der anonymen Funktion als Faktoren zur Berechnung der X-Koordinate des zweiten Punktes verwendet.

```
1(defn example []
2  (render
3    (compose 800 2000 {:antialiasing :on}
4        (with-attributes {:paint :red}
5          (dotimes [n 10]
6              (doall (map
7                      (fn [val](if (= 0 (mod n 2))
8                              (line 400 50 (* n val) 1000)
9                              (line 400 50 (* n val) 1000 {:paint :blue})))
10                      (range 100 106)))))))
```

Listing 5.10 Beispiel Clojure Funktionalität

[19] Werte einer Collection werden erst erzeugt, wenn diese benötigt werden. Auf diesem Wege sind unendlich große Collections möglich.

So ergeben sich exemplarische im letzten Durchlauf die Koordinaten für die Linien und das Bild dargestellt in Abbildung 5.1:

x(400 50), y((9*100) 1000))

x(400 50), y((9*101) 1000))

x(400 50), y((9*102) 1000))

x(400 50), y((9*103) 1000))

x(400 50), y((9*104) 1000))

x(400 50), y((9*105) 1000))

Deutlich wird, dass sich die Abstände der Linien innerhalb eines Linienblockes mit steigendem Wert *n* vergrößern. Ein Block besteht aus sechs Linien mit derselben Farbe, da alle denselben Faktor nutzen.

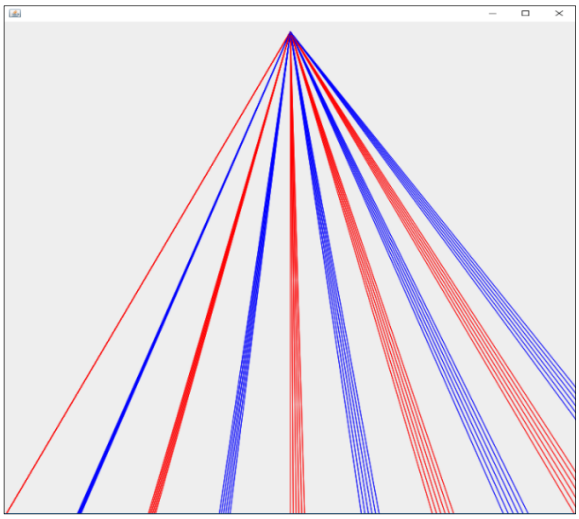

Abbildung 5.1 Beispiel Clojure Funktionalität

5.2.10. Veröffentlichung

Image-Compojure ist auf der Plattform Clojars[20] publiziert. Clojars ist ein von einer Community gepflegtes Repository für Open Source Bibliotheken. Über die Programme Leiningen, Maven oder Gradle kann die DSL in ein neues Projekt eingebunden werden. Zu diesem Zweck muss beispielsweise im Falle eines Leiningen Projekts, der Projektdatei zu den *dependencies* der Eintrag *[image-compojure "1.0.0"]* hinzugefügt werden. Durch die Angabe von *(:require [image-compojure.core :as compojure])* in der Namespace-Definition einer Datei stehen diesem Namespace alle Funktionen zur Verfügung.

[20] https://clojars.org/

5.3. Design und Implementierung

In diesem Teil der Arbeit geht es um die Konzeptionierung des Konstrukts der Sprache, sowie dessen Implementierung. Zunächst werden der logische Aufbau und gleichzeitig einzelne Makros und Funktionen mit zugehörigen Quellcode-Passagen analysiert[21].

5.3.1. Konstrukt

Die Basis des Konstrukts stellen vier Variablen des Typs Var, im Folgenden als Default-Vars bezeichnet, dar. An sie ist jeweils ein Initialwert gebunden. Die Default-Vars sind als ^:*dynamic* definiert und können somit später neue Referenzen erhalten. Eingangs sind diese wie folgt definiert:

- ✓ ***default-image*** : Wert: BufferedImage mit Größe 1920x1080
- ✓ ***default-g2d*** : Wert: G2D-Objekt erstellt aus *default-image*
- ✓ ***default-render-settings*** : Wert: Map – definiert alle Standardwerte der Render-Settings
- ✓ ***default-shape-attributes*** : Wert: Map – definiert alle Standardwerte der Shape-Attribute

Das Var *default-image* beinhaltet das Bild, welches am Ende eines Compositing-Prozesses durch die *render* Funktion ausgegeben werden kann. Aus diesem Bild wird ein G2D-Objekt erstellt und an *default-g2d* gebunden. Dieses Objekt stellt den bereits erwähnten Grafikkontext bereit, in dem gezeichnet wird. Alle Rendering-Operationen und Angaben von Shape-Attributen werden auf *default-g2d* angewendet. Werden Attribute übergeben, zum Beispiel durch das *with-attributes* Makro, hat dies eine Veränderung des Zustands von *default-g2d* zur Folge. Diese Zustandsmanipulation bleibt vor dem Entwickler verborgen, da Image-Compojure die dafür zuständigen Funktionen, wie beispielsweise *set-stroke* selbstständig ausführt und anschließend rückgängig macht. Auf diesen Punkt wird im späteren Verlauf noch einmal Bezug genommen. Wie die Zustandsmanipulationen sind auch die vier Default-Vars für den Entwickler nicht relevant bzw. unsichtbar. Der Entwickler hat theoretisch Zugriff auf die Vars, jedoch sind diese Kenntnisse nicht erforderlich. Die Empfehlung lautet hierbei, den direkten Zugriff auf diese zu vermeiden. Über die Verwendung der Makros *compose* und *with-attributes* arbeitet das Programm intern mit den Vars.

Beim Aufruf von *compose* werden neue Werte an die Vars *default-image* und *default-g2d* gebunden. Diese Werte, von nun an als Bindings bezeichnet, existieren nur in dem Kontext des derzeitigen Makro-Aufrufs.

[21] Der gesamte Code kann im Repository eingesehen werden: https://github.com/NicolasSch/image-compose

Alle Rendering-Operationen werden auf die derzeitigen Bindings angewendet. Die Wahl der Managed Reference fiel auf den Typ Var, da es bei gleichzeitigem Zugriff auf das Symbol irrelevant ist, welche Werte diesem derzeitig in anderen Threads zugeordnet sind. Durch diese Implementierung des *compose* Makros wird Threadsicherheit für multiple Threads erreicht. Wird das compose Makro nicht verwendet, kann die Threadsicherheit nicht gewährleistet werden. Wenden viele Threads gleichzeitig Rendering-Operationen ohne *compose* an, werden all diese auf dasselbe Objekt angewendet, referenziert durch *default-g2d*. Nutzt der Anwender *compose* in jedem Thread, operieren all diese jeweils auf ihrem eigenen G2D-Objekt.

Compose kann ebenfalls einen neuen Wert an das Var *default-render-settings* binden. Hierzu wird die vorhandene Map in *default-render-settings* mit der als Argument übergebenen Map verschmolzen. Es entsteht so eine neue Map mit neuen Werten. Alle Attribute, die nicht angegeben wurden, bleiben erhalten. *With-attributes* funktionieren auf die gleiche Art, jedoch mit dem Unterschied, dass hier nur das Var *default-shape-attributes* neu gebunden wird. Da sich die beiden Makros sehr ähnlich sind, wird nur ersteres mit Hilfe des Listings 5.11 näher betrachtet.

```
1 (defmacro compose
2   [w h & forms]
3   (loop [settings (first forms)
4          body (next forms)]
5     (if (map? settings)
6       `(let [image# (BufferedImage. ~w ~h BufferedImage/TYPE_INT_ARGB)]
7          (binding [default-image image#
8                    default-g2d (.createGraphics image#)
9                   default-render-settings ~(merge default-render-settings
10                                                  settings)]
11           (set-render-settings default-render-settings)
12           (do
13             ~@body
14             default-image)))
15       (recur {} forms))))
```

Listing 5.11 Implementierung compose

Zunächst können zwei Argumente zum Bestimmen der Breite und Höhe des Bildes, sowie eine beliebige Anzahl an Ausdrücken übergeben werden. Daraufhin überprüft der If-Block, ob es sich bei dem ersten Element in der Liste *forms* um eine Map handelt. Dazu wird dieser Wert durch die Funktion *loop* an das Symbol *settings* gebunden. Handelt es sich nicht um eine Map, erfolgt ein rekursiver Aufruf über die Funktionen *recur* mit einer leeren Map als erstes und einer Liste bzw. S-Expression, die alle weiteren Ausdrücke enthält, als zweites Argument. Im darauffolgenden Let-Block wird eine neues BufferedImage erstellt und an *image* gebunden. Im Anschluss findet das soeben erläuterte Binding der Default-Vars statt. Durch die Funktion *createGraphics* wird das neue G2D-Objekt erstellt. Mit Hilfe der Funktion *merge* werden die *settings* mit den *default-render-settings* zusammenführt. Die Ergebnisse werden neu an die Vars

gebunden. Nun wird die neue Map an die Funktion *set-render-settings* übergeben und dort auf das Objekt in *default-g2d* angewendet. *Set-render-settings* erstellt hierzu eine neue Map mit den Keys und Values, die in Java zum Setzen der Rendering Hints benötigt werden. Schließlich werden alle Ausdrücke ausgeführt, die dem Symbol *body* zugewiesen sind.

Das Makro *with-attributes* ist weniger komplex als *compose*, denn hier wird nur das angesprochene Bindung vorgenommen und anschließend die Funktion *set-shape-attributes* aufgerufen. Sind alle Rendering-Operationen abgeschlossen, wird *set-shape-attributes* erneut aufgerufen, um *default-g2d* wieder in den vorherigen Zustand zu versetzen. Die Funktion s*et-shape-attributes* überprüft, welche Attribute manipuliert werden müssen und setzt diese anschließend.

Ähnlich funktioniert auch das *with-transform* Makro (Listing 5.12). Dieses nimmt ein AffineTransform-Objekt entgegen, konkateniert dieses mit der derzeitigen Transformation und setzt zum Schluss die Transformation wieder zurück. Bei den Funktionen *concat-transform* und *set-transform* handelt es sich um Wrapper-Funktionen, die den Java Interop vornehmen.

```
1 (defMakro with-transform
2   ([trans & forms]
3    `(let [current-trans# (.getTransform default-g2d)
4           new-trans# (concat-transforms (.getTransform default-g2d)~trans)]
5       (do
6         (.transform default-g2d new-trans#)
7         ~@forms
9         (set-transform current-trans#))
10        )))
```

Listing 5.12 Implementierung with-transform

5.3.2. Maps, Keys und Values

Die Selektion der Werte zu Attributen eines Shape-Objekts, der Render-Settings oder Schriftzügen, findet nur über Keys statt. Ausnahmen bilden lediglich Attribute, die einen nummerischen Datentyp oder ein Objekt der Klasse *java.awt.Color* erwarten. Der Grund hierfür ist, dass die Attributwerte aus Java, die den Zustand des G2D-Objekts bestimmen, in Maps definiert sind. Durch die Verwendung von Keys kann auf diese Werte direkt zugegriffen werden. So gibt beispielsweise der Ausdruck *(:plain font-styles)* direkt den Wert *Font/PLAIN* aus der Map *font-styles* zurück. Strings oder Integer Werte wären ebenfalls möglich gewesen, dies hätte jedoch die Komplexität des Programms gesteigert. Um eine Auswahl zu treffen, wären viele If- oder Cond-Blöcke notwendig gewesen. Ein Cond-Block findet zum Beispiel in der Funktion *color* Verwendung. Außerdem entsprechen Keys eher dem subjektiven Programmierstil, der in Clojure bevorzugt wird.

5.3.3. Funktionen für Shapes

Die Funktionen zum Zeichnen von Shapes nutzen die Funktion höherer Ordnung *draw-fill*, um festzustellen, ob eine Shape-Objekt ausgefüllt oder lediglich umrandet werden soll. Zu diesem Zweck werden zwei anonyme Funktionen definiert und der Funktion *draw-fill* zusammen mit einer Map zur Bestimmung der Shape-Attribute übergeben. Die Funktion, die als zweites Argument übergeben wird, zeichnet eine Umrandung, die nachfolgende Funktion eine ausgefüllte Form. Die anonymen Funktionen werden *draw-fill* übergeben, sodass nicht jede Funktion, die ein Shape-Objekt zeichnet, diese Prüfung vornehmen muss. Draw-fill wiederum verwendet das Makro *draw-fill-reset*, welches prüft, ob Elemente in der Map *attributes*, welche die Shape-Attribute angibt, vorhanden sind. Sind Einträge vorhanden, werden die entsprechenden Attribute durch die Funktion *set-shape-attributes* gesetzt und nach Abschluss des Zeichnens durch den Aufruf von *reset-shape-attributes* wieder zurückgesetzt.

Draw-fill und *draw-fill-reset* besitzen die gleiche Aufgabe. Beide prüfen einen Wert auf logische Wahrheit und führen dem Ergebnis entsprechend Funktionen aus. Es ist ebenfalls möglich, *draw-fill* als ein Makro zu implementieren. Hierdurch kann erreicht werden, dass keine anonymen Funktionen mehr übergeben werden müssen. An dieser Stelle wurde bewusst der Weg über Closures gewählt, um zu verdeutlichen, dass die Evaluation von Funktionen auch mit anderen Mitteln als Makros aufgeschoben werden kann (vgl. Kapitel 3.7).

```
1 (defn draw-fill
2       [attributes fill-func draw-func]
3   (let [fill (:fill attributes)
4         attributes (dissoc attributes :fill)]
5     (draw-fill-reset attributes
6                      (if fill
7                        (fill-func)
8                        (draw-func)))))
```

Listing 5.13 Implementierung draw-fill

In Listing 5.13. ist die Implementierung der *draw-fill* Funktion abgebildet. Hier wird zunächst der Key :fill aus *attributes* entfernt und anschließend *draw-fill-reset* aufgerufen. Ersichtlich wird, dass der If-Block nicht mehr als anonyme Funktion übergeben werden muss, um festzustellen, ob das Argument *fill-func* oder *draw-func* auszuführen ist, da es sich bei *draw-fill-reset* um ein Makro handelt. Man beachte, dass nicht die einzelnen Funktionen übergeben werden, sondern der komplette If-Block. Für *draw-fill-reset* ist nur der Inhalt der Map *attributes* entscheidend. Die übergebene S-Expression wird vom Makro wieder zurückgegeben und anschließend im Kontext der *draw-fill* Funktion evaluiert.

Exemplarisch für alle Funktionen zum Zeichnen von Shapes wird nun die Funktion *polygon* beschrieben (siehe Listing 5.14). Dadurch wird deutlich, wie *draw-fill* verwendet wird. Dieser Funktion werden die beiden anonymen Funktionen in Zeile 7 und 8/9, sowie die Attribute übergeben. Erstere füllt die Form, letztere zeichnet nur die Umrandung. *Polygon* kann auch ohne Shape-Settings als Argument aufgerufen werden. In diesem Fall ruft *polygon* sich selbst auf. Hierbei übergibt die Funktion die eingegeben Argumente x y und zusätzlich eine neue Map, die das Attribut *:fill* beinhaltet. Auf diese Weise werden Shape-Objekte standardmäßig nur umrandet. Die Funktion beinhaltet eine Fehlererkennung in Form einer Runtime-Exception. Diese prüft, ob die Anzahl der Werte zur Bestimmung der X- und Y-Koordinaten gleich sind. Diesem Schema folgen alle Funktionen dieser Art.

```
1 (defn polygon
2  ([x y attributes]
3   (if-not (= (count x) (count y))
4      (throw (RuntimeException.
5                    "Number of x and y coordinates must be equal")))
6       (draw-fill attributes
7            #(.fillPolygon default-g2d x y (count x))
8            #(.drawPolygon default-g2d (int-array x) (int-array y)
9              (count x))))
10
11  ([x y]
12   (polygon x y {:fill false})))
```

Listing 5.14 Implementierung polygon

5.3.4. Schriftzüge

Im Schlussteil dieses Abschnitts wird die Implementierung von Schriftzügen erläutert. Es existieren zum Zeichnen von Schriftzügen zwei Funktionen. Wie bereits erklärt: Die erste Funktion *styled-text* erwartet eine Map, die einen Schriftzug und einen Schriftstil beinhaltet. Diese Map kann über die zweite Funktion *create-styled-text* durch die Angabe von verschiedenen Argumenten zur Attributbeschreibung[22] erstellt werden. Styled-text setzt nun die Attribute für die übergebene Schriftart und zeichnet anschließend den Text. Dabei merkt sich die Funktion vorherige Attribute und setzt diese nach Abschluss der Rendering-Operation wieder zurück.

```
1 (defn create-styled-text
2   ([text name style size {:keys [weight width underline foreground
3        background strike-through swap-colors kerning posture]}]
4    (let [font (create-font name style size)
5          styled-font (when-> {}
6                              kerning (assoc (TextAttribute/KERNING)
7                                             (TextAttribute/KERNING_ON))
8                              posture (assoc (TextAttribute/POSTURE)
9                                             (posture text-posture))
10      …
11      ))]
12    (-> {}
13        (assoc :text text)
14        (assoc :font (.deriveFont font styled-font)))))
```

Listing 5.15 Implementierung create-styled-text

Das Listing 5.15 zeigt der Übersichtlichkeit halber die gekürzte Implementierung der *create-styled-text* Funktion. Hierbei kommen zwei Makros zum Einsatz - Zum einen das normale Thread-First -> und die Modifizierung von diesem, das when-> Makro. Beide Makros funktionieren auf die Weise, dass die initial übergebene Datenstruktur als erstes Argument den nachfolgenden Funktionen übergeben wird. In diesen Fällen werden zwei leere Maps als Eingangsdatenstrukturen übergeben. Die Funktionen reichen dabei immer die neu entstandene Datenstruktur an die nächste Funktion weiter (Method-Chaining). Das eigens modifizierte when-> Makro prüft zusätzlich eine Bedingung auf Wahrheit. Wird die Bedingung erfüllt, wird der nachfolgende Ausdruck zurückgegeben. Ebenfalls findet hier Destructuring zur Auswahl der Java Konstanten für TextAttributes Verwendung. Durch diese können bestehende Schriftarten individuell angepasst werden. Durch das Destructuring werden die Values der Keys aus der Map, die an *create-style-text* übergeben wird, an gleichnamige Symbols gebunden. Sind Keys nicht angegeben, ist der Wert des Symbols *nil*. Die Symbols werden nun an *when->*, gefolgt von dem Ausdruck *(assoc Key Value)*, übergeben. Vereinfacht ausgedrückt bedeutet es, dass das Makro prüft, ob dem Symbol ein Wert zugewiesen wurde. Ist

[22] Die verfügbaren Attribute können in der Tabelle 5.3 „Attribute für Schrift“ eingesehen werden (Material A.4).

dies der Fall, erstellt *assoc* eine neue Map, die alle vorherigen Werte zuzüglich der neuen Werte beinhaltet. Auf diese Weise müssen keine If- oder Cond-Blöcke verwendet werden. Der Wert des Symbols ist wiederum ebenfalls ein Key, der die entsprechende Konstanten aus den definierten Maps *text-weight, text-width, text-postur* oder *text-underline* zurückgibt. Das Thread-first Makro -> erstellt eine neue Schriftart aus der im Let-Block erstellten Schriftart und der, durch das *when->* Makro definierten Map, welche die TextAttributes beinhaltet. Es speichert diese neue Schriftart zusammen mit dem darzustellenden Text in einer neuen Map und gibt diese anschließend zurück.

5.4. Ausblick

In diesem Teil wird dargelegt, für welchen ersten realen Anwendungsfall die DSL verwendet wird. Hierzu wird kurz der Anwendungsfall dargestellt und anschließend beschrieben, welche Aufgabe dabei Image-Compojure übernimmt und wie das Endergebnis aussehen wird. Diesem geht eine Beschreibung der geplanten Weiterentwicklung voraus, die sich mit der Frage beschäftigt, welche Features als nächstes umgesetzt werden.

5.4.1. Weiterentwicklung

Im nächsten Schritt werden der DSL Runtime-Exceptiones hinzugefügt, um Entwicklern aussagekräftige Fehlermeldungen beim Programmieren zu liefern. Vereinzelt sind bereits Exceptiones vorhanden, jedoch in einem sehr geringen Umfang und lediglich für triviale Anwendungsfälle, wie zum Beispiel in der Funktion *polygon*, welche die Anzahl der X- und Y-Koordinaten auf Gleichheit prüft. Um die Weiterentwicklung der DSL selbst zu vereinfachen, sind Modultests geplant, die einzelne Funktionen der Sprache automatisch testen. Modultest nehmen Eingabewerte und das erwartetet Ergebnis entgegen. Sollte es während des Prüfprozesses zu Fehlern kommen, werden die entsprechenden Funktionen kenntlich gemacht. Modultests sind in diesem Fall nicht trivial, da der Großteil der DSL auf Funktionen mit Nebeneffekten beruht, welche durch Zustandsmanipulationen die Eigenschaften eines Objekts verändern. Ebenfalls ist problematisch, dass für neu kreierte Bilder, Ergebnisbilder vorhanden sein müssen. Zurzeit existiert nur eine nicht vollständig abgedeckte Qualitätssicherung durch automatisierte Tests, deren Ergebnisse manuell geprüft werden müssen. Neben diese Weiterentwicklungen ist auch eine Erweiterung des Funktionsumfangs geplant. So soll es zukünftig möglich sein, Bilder mit unterschiedlichen Algorithmen zu schärfen oder unscharf darzustellen.

5.4.2. Verwendung bei AnyUp

Die fertige DSL wird in der Firma AnyUp in Verbindung mit Facebooks Open Graph Meldungen verwendet. Open Graph Meldungen ermöglichen es Benutzern einer Smartphone-App, Statusmeldungen mit Bildern, Texten oder Standorten auf Facebook zu veröffentlichen (Post). Hinter diesem Post verbirgt sich ein Link, der beispielweise auf eine externe Webseite weiterleiten kann. Für den vorliegenden Fall soll es einem Benutzer möglich sein, nach Absolvierung eines Workouts, das erzielte Ergebnis mit Freunden auf Facebook zu teilen. Zu diesem Zweck kann der Benutzer ein Bild aufnehmen. Nimmt er keins auf, wird ein Standardbild geladen. Dieses wird anschließend mit den Informationen bezüglich des Workout-Inhalts, dem Namen des Benutzers und dem erreichten Ergebnis in einem neuen Bild dargestellt. Der Link des Bildes verweist auf die Webseite der Firma AnyUp[23]. Die Komposition wird serverseitig mit Image-Compojure vorgenommen. Das Ergebnis einer solchen Komposition, das per Open Graph Meldung auf Facebook geteilt wird, ist in Abbildung 5.2 dargestellt (für den Quellcode siehe Material A.4, nachfolgende Zeilenangaben sind hierauf bezugnehmend).

Abbildung 5.2 AnyUp Open Graph

[23] http://www.anyup.net/

In diesem Bild findet ein Großteil der DSL Features Anwendung, so zum Beispiel das Laden von Bildern (Z. 2, 6), die Definition von verschiedenen Schriftarten (Z. 9-15), das Erstellen von Farbobjekten (Z. 7), das Zeichnen von Linien und einem Shape-Objekt, das sowohl ausgefüllt, als auch umrandet gezeichnet wird (Z. 40, 53 und 24-25), und das Erstellen sowie Anwenden von Transformationen (Z. 26 - 33). Die Transformationen werden hier benutzt um die gezeichneten Texte in Blöcken im Bild zu positionieren (Z. 31-55). Die angegebenen Strings sind derzeitig hart codiert, sollen jedoch in der späteren Anwendung dynamisch befüllt werden.

6. Fazit und Zusammenfassung

Im Rahmen dieser Arbeit wurde erfolgreich eine interne domänenspezifische Sprache in Clojure realisiert, welche die Java 2D API nutzt, um Bilder parallel in multiplen Threads zu komponieren. Diese DSL wurde auf der Plattform Clojars veröffentlicht und steht der Open Source Community zur Verfügung. Eine Beispielanwendung, welche diese DSL verwendet, ist implementiert und ebenfalls auf der Plattform GitHub[24] veröffentlicht worden.

Die Sprache Clojure war für die Implementierung dieser DSL äußerst komfortabel. Dies lag zum einen an der Java Interoperabilität und die hierfür bereitgestellten Makros (vgl. Kapitel 3.5 Java Interoperabilität). Die Verwendung von Java Klassen sowie deren Funktionen und Variablen gestaltete sich als sehr effizient und angenehm. Das ist unter anderem dem Quellcode zu verdanken, der sich zu großen Teilen übersichtlicher und kürzer implementieren ließ, als es in Java möglich gewesen wäre. Zum anderen trug das Makro-System erheblich zur komfortablen Implementierung bei. Durch dieses System war es möglich, nacheinander ausgeführte Ausdrücke miteinander zu verbinden und so ein anderes Ergebnis zu erreichen, als die Ausdrücke für sich allein erzielt hätten. Die in der Arbeit konzipierte DSL besitzt eine Grammatik, mit der aus einzelnen Funktionen (Objekte und Prädikate), Makros (Adverbiale) und Attributen in Form von Maps (Prädikative Glieder) komplexe Sätze konstruiert werden können. Das Subjekt stellt das *compose* Makro und das darin erstellte Bild dar. Ohne dieses funktioniert ein Satz und somit auch die DSL nicht. Durch Funktionen (Shapes, Bilder oder Schriften) wird beschrieben, was einem Subjekt (Bild) in einem Satz (*with-attributes* oder *with-transform*) widerfährt. Während durch Shape-Attribute definiert wird, in welcher Gestalt (Farben etc.) sich das Bild letztendlich darstellt. Makros fügen die Sätze zu einem Text beziehungsweise zu einer Bildkomposition zusammen.

Die eingangs gestellte Frage, ob sich die Implementierung von Multithreading durch die Verwendung einer funktionalen Sprache einfacher als in OO Sprachen gestalten würde, hat sich eindeutig bestätigt. Der Einsatz im Rahmen eines Multithreadingszenarios ist durch die Verwendung der Managed References ausreichend abgesichert. Es musste lediglich entschieden werden, welcher Typ der Managed References in dem vorliegenden Fall am geeignetsten ist. Die Entscheidung fiel auf den Typ Var (vgl. 5.3.1 Konstrukt).

Rückblickend lässt sich sagen, dass es sich beim Design und der Absteckung der spezifischen Domäne um einen ungeahnt anspruchsvollen Teil dieser Arbeit handelte. Im Normalfall wird eine Domäne von einem sogenannten Domänen-Experten bearbeitet. Dieser weiß, welche Aufgaben er zu lösen hat und wie er diese

[24] https://github.com/NicolasSch/example-image-compojure

bewältigen kann. So kann der Domänen-Experte einem Entwickler ganz genau vorgeben, welche Funktionalitäten die DSL abdecken muss. Der Entwickler läuft somit nicht Gefahr, die Sprache unnötig komplex zu gestalten. Hinzu kommt, dass ein Entwickler auf Basis der bisherigen Lösungswege neue, unkompliziertere und kürzere Wege konzeptionieren und diese in Zusammenarbeit mit dem Domänen-Experten weiterentwickeln kann. Auf diese Weise wird gewährleistet, dass die neue Sprache nicht komplizierter wird, als der bereits existierende Lösungsweg ist. Wie auch im vorliegenden Fall kann Experte und Entwickler durch dieselbe Person beschrieben werden. Zu Beginn der Arbeit stand die Entscheidung, welche Features neben den von AnyUp stammenden Vorgaben sinnvolle Erweiterungen zur DSL darstellen. Hierzu musste zunächst festgestellt werden, welches Spektrum die Java 2D API überhaupt abdeckt. Es bestand die Herausforderung, ansatzweise selbst zu einem Domänen-Experten zu werden. Zu diesem Zweck erfolgte eine eingehende Analyse der grundlegenden Funktionsweise der API. Weiterhin musste festgestellt werden, wie eine Bildkomposition bisher in Java erfolgt und welche Vor- und Nachteile dabei bestehen. Anschließend ging es um die Entwicklung einer eigenen Syntax, die kürzer und bei weitem weniger Komplex ist als bisher in Java. Dies wurde durch die Verschleierung der Zustandsmanipulationen, die durch die 2D API vorgenommen werden, erreicht. Diese sind in der DSL komplett durch die Verwendung des *with-attributes* Makros mit einer Map zur Festlegung der Shape-Attribute, sowie Funktionen, die eine solche Map als Argument entgegennehmen, abgedeckt (vgl. Kapitel 5.2.2. Shapes und 5.2.4 with-attributes). Der Entwickler ist nicht länger gezwungen, den Zustand globaler Variablen manuell zu steuern. Dies wird durch die Angabe der Keys in den Maps programmatisch vorgenommen. Angesichts dieser Tatsache wird eine weitaus geringere Fehlerempfindlichkeit als zuvor erreicht. Während des Entwicklungsprozesses kann sich der Programmierer nun voll und ganz auf die Komposition des Bildes mit seinen vielen möglichen Bestandteilen konzentrieren. Dieser muss sich nicht länger mit Eigenheiten der 2D API beschäftigen, die für eine Komposition selbst unerheblich sind. Das Ziel der Arbeit wurde somit erreicht.

A. Material

A.1 Quellcode Composite Beispiel

```
(render-output
  (compose 1000 600
         {:antialiasing      :on
          :text-antialiasing :on}
         (rectangle 50 100 200 200 {:paint :black :fill true})
         (oval 50 50 200 200 {:paint (color 255 0 0 255) :fill true :composite
                                                                 :src_over})

         (rectangle 350 100 200 200 {:paint :black :fill true})
         (oval 350 50 200 200 {:paint (color 255 0 0 125) :fill true :composite
                                                                 :src_in})

         (rectangle 650 100 200 200 {:paint :black :fill true})
         (oval 650 100 200 200 {:paint (color 255 0 0 255) :fill true :composite
                                                                 :dst_out})

         (rectangle 50 400 200 200 {:paint :black :fill true})
         (oval 50 350 200 200 {:paint (color 255 0 0 255) :fill true :composite
                                                                 :src_out})

         (styled-text 25 75 (create-styled-text "A" :plain :bold 30 {:foreground
                                                                 :yellow}))
         (styled-text 325 75 (create-styled-text "B" :plain :bold 30 {:foreground
                                                                 :black}))
         (styled-text 25 350 (create-styled-text "D" :plain :bold 30 {:foreground
                                                                 :black}))
         (styled-text 625 75 (create-styled-text "C" :plain :bold 30 {:foreground
                                                                 :black}))
         (styled-text 325 350 (create-styled-text "E" :plain :bold 30 {:foreground
                                                                 :black}))

         (styled-text 110 90  (create-styled-text "src_over"))
         (styled-text 420 90  (create-styled-text "src_in"))
         (styled-text 110 390 (create-styled-text "src_out"))
         (styled-text 720 90  (create-styled-text "dst_out"))

         (rectangle 350 400 500 200 {:paint :red :fill true})
         (rectangle 550 350 100 200 {:paint :green :fill true :xor-mode
                                                                 :blue})))
```

Listing A.1 Composite Beispiel Quellcode

A.2 Erstellen von Paint Values

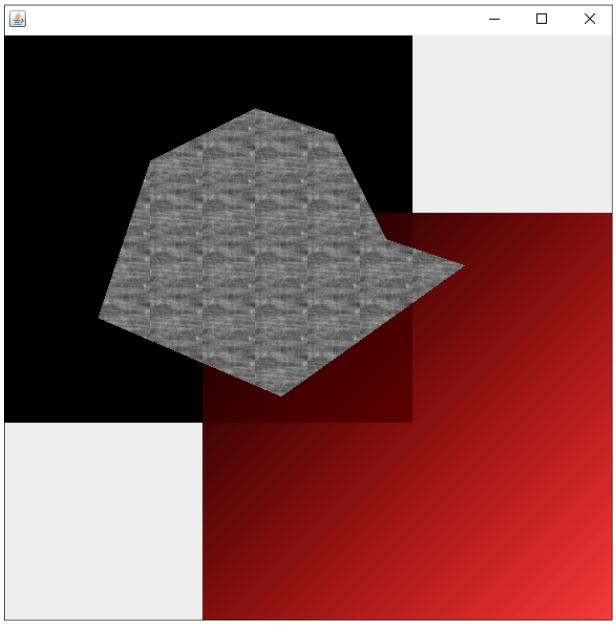

Abbildung A.1 Erstellung von Paint Values

A.3 Beispiel von simplen Zeicheinoperationen

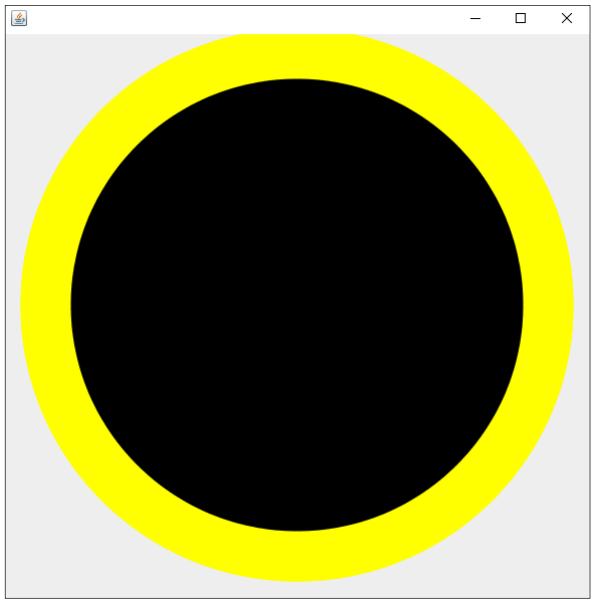

Abbildung A.2 Beispiel simpler Rendering-Operationen

A.4. Attribute für Schriftzüge

Attribut	Arg/ Key	Value	Beschreibung
Schriftart	name	:dialog :dialog-input :sans-serif :serif :momospaced :times	Dient zur Auswahl der Schriftart. Es stehen alle logischen Schriftarten zur Verfügung. Physikalische Schriftarten können hinzugefügt werden.
Style	style	:plain :bold :italic :roman-baseline :center-baseline :hanging-baseline :true-type-font :type1-font	Gibt den Style der Schriftart an. Zur Verfügung stehen alle in Java verfügbaren Styles. Styles sind vordefinierte Werte für die nachfolgenden Attribute, ausgenommen der Größe.
Größe	size	float	Definiert die Größe der Schrift.
Unterstrich	:underline	:low-on-pixel :low-two-pixel :low-dotted :low-gray :low-dashed	Sofern angegeben wird der Text mit einem Unterstrich gerendert. Die Values definieren das Aussehen der Linie.
Schriftfarbe Hintergund	:foreground :background	Color-Key oder Color Color-Key oder Color	Setzt die Farbe der Schrift oder die des Hintergrundes. Es kann direkt ein Key zur Auswahl der Farbe oder ein java.awt.Color Objekt übergeben werden.
Breite	:width	:condensed :semi-condensed :regular :extended :semi-extended	Gibt die Breite des zu Schriftzuges an. Zur Verfügung stehen alle in Java verfügbaren Werte.
Stärke	:weight	:extra-light :light :demilight :regular :medium :semibold :demibold :bold :heavy :extrabold :ultrabold	Hiermit wird Stärke der einzelnen Glyphen oder auch Buchstaben des Schriftzuges bestimmt. Anders als die Breite bezieht sich die Stärke nur auf eine einzelne Glyphe.
Neigung	:posture	:regular :onlique	Bestimmt die Neigung der einzelnen Glyphen. Verwendet werden können die in Java definierten Werte.
Abstand	:kerning	boolean	Anpassung des Abstands zwischen Glyphen zueinander
Gestrichen	:strike-through	boolean	Bei logischer Wahrheit, wird der Schriftzug durchgestrichen.
Farbtausch	:swap-colors	boolean	Vertauscht :foreground mit :background.

Tabelle A.1 Attribute für Schrift

A.5. Abbildung zur Verwendung create-styled-text

Abbildung A.3 Verwendung create-styled-text

A.6. Quellcode zur Abbildung AnyUp Open Graph

```
(defn test-comp []
  (let [img (load-image "res/bg-1.JPG")
        w (.getWidth img)
        h (.getHeight img)
        logo #(image 0 0
               (load-image "res/Logo-white.png"))
        anyup-red (color 233 80 65 255)
        headline (create-styled-text "Headline"
                  :sans-serif :italic 50 foreground (color :white)})
        wod (create-styled-text "wod-description"
               :sans-serif :bold 25 {:foreground (color :white)})
        name (create-styled-text "score"
               :sans-serif :italic 30 {:foreground (color :white)})
        date (create-styled-text "score" :sans-serif :plain 20
               {:foreground (color :white)})
        black (color 0 0 0 200)
        polygon (Polygon. (int-array [0, 0, w, w, (/ w 3), (/ w 3)])
                          (int-array [0, h, h, (* h 0.65), (* h 0.85), 0]) 6)]

    (render
      (compose
        (.getWidth img) (.getHeight img) {:antialiasing :on}
        (image 0 0 img)
        (shapes [polygon] {:fill true :paint black})
        (shapes [polygon] {:paint anyup-red :width 8})
        (with-transform
                       (transform
                                (translate (* w 0.5) (* h 0.8))
                                (rotate 0))
                         (logo))
        (with-transform
          (transform
            (translate 0 200))

          ;-------------Text Name ------------------------------

          (styled-text 50 0 (merge name {:text "Nicolas Schwartau"}))
          (styled-text 100 30 (merge date {:text "Sonntag, 08.05.2016"}))

          (line 50 50 (- (/ w 3) 50) 50 {:paint anyup-red})

          ;--------------Text Wod Description--------------

        (styled-text 50 200 (merge headline {:text "Workout: DAWN"}))
        (styled-text 100 250 (merge wod {:text "Auf Zeit:"}))
        (styled-text 100 300 (merge wod {:text "50  Squats"}))
        (styled-text 100 335 (merge wod {:text "40  Sit-Ups"}))
        (styled-text 100 370 (merge wod {:text "30  Push-Ups"}))
        (styled-text 100 405 (merge wod {:text "10  Pull-Ups"}))

        ;----------------Text Score--------------

        (line 50 600 (- (/ w 3) 50) 600 {:paint anyup-red})
        (styled-text 50 700 (merge wod {:text "Dein Ergebnis:"}))
        (styled-text 100 800 (merge headline {:text "5:33 Minuten"})))))))
```

Listing A.2 AnyUp Open Graph Quellcode

Literaturverzeichnis

Buchquellen

Fowler (2010)

Fowler, M. Domain Specific Language, September 24, 2010, Addison-Wesley Professional

Rathore / Avila (2016)

Rathore,A. und Avila, F., Clojure in Action, 2016, Second Edition, Manning Publications Co.

Higginbotham (2015)

Higginbotham, D, Clojure for the Brave and True,

Kapitel: Clojure Alchemy: Reading, Evaluation, and Makros, Absatz: Special Form

Online-Version: http://www.braveclojure.com/read-and-eval/

Kapitel: Core Functions in Depth, Absatz: Programming to Abstractions)

Online-Version: http://www.braveclojure.com/core-functions-in-depth/

Knuden (1999)

Kunden, J. JAVA 2d Graphics,1999, Verlag O`Reilly

Voelter (2010-2013)

Voelter, M., DSL Engineering, Designing, Implementing and Using Domain-Specific Languages, Online-Version http://dslbook.org/

Internetquellen

Facebook Developer

Zugriff: 22.05.2016

https://developers.facebook.com/docs/sharing/opengraph

Innoq

Zugriff: 04.05.2016

https://www.innoq.com/de/articles/2012/07/domain-specific-languages/

Oracle JAVA docs

Overview of the Java 2D API Concepts, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/overview/index.html

Coordinates, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/overview/coordinate.html

Class Graphics, Zugriff: 22.05.2016

https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics.html

Class Graphics2D, Zugriff: 22.05.2016

https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics2D.html

Java 2D Rendering, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/overview/rendering.html

Controlling Rendering Quality, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/advanced/quality.html

Geometric Primitives, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/overview/primitives.html

Stroking and Filling Graphics Primitives, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/geometry/strokeandfill.html

Physical and Logical Fonts, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/text/fonts.html

Measuring Text, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/text/measuringtext.html

Using Text Attributes to Style Text, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/text/textattributes.html

Compositing Graphics, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/advanced/compositing.html

Transforming Shapes, Text, and Images, Zugriff: 22.05.2016

https://docs.oracle.com/javase/tutorial/2d/advanced/transforming.html

Class AffineTransform, Zugriff: 22.05.2016

https://docs.oracle.com/javase/8/docs/api/java/awt/geom/AffineTransform.html

IT Wissen (23.05.2016)

Threads, Zugriff: 16.05.2016

http://www.itwissen.info/definition/lexikon/Thread-thread.html

Rich Heckey (2012)

Präsentation Boston Lisp Meeting 2012
https://www.youtube.com/watch?v=7mbcYxHO0nM&ct=00h21m04s

Eigenständigkeitserklärung

Ich versichere, die vorliegende Arbeit selbständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

(Ort, Datum) (Nicolas Schwartau)