Projektarbeit

Reflection-basierte Genetische Programmierung am Beispiel Evolutionärer Kunst

An der Fachhochschule Dortmund
im Fachbereich Informatik
Studiengang Informatik
Vertiefungsrichtung Praktische Informatik
erstellte Projektarbeit

von
Johannes Rückert

Betreuer:
Prof. Dr.-Ing. Christoph M. Friedrich

Dortmund, 16. September 2013

Inhaltsverzeichnis

Abbildungsverzeichnis

Abstract

Genetic Programming allows for the automatic solving of problems using a given function set, from which individuals are generated, evaluated, selected, mutated and recombined based on the principles of natural evolution. *Reflection* enables a program to know and modify its own structure. In the case of object-oriented programming languages this also includes the ability to read meta information about classes and objects at runtime. Evolutionary Art is an area of Artificial Art where images are created and evolved, using, for example, Genetic Programming. The most important element of a GP system is its function set, because it defines the solution space. Different frameworks require different kinds of function sets, which in most cases requires the function set to be reimplemented when used in different applications. *Reflection-based Genetic Programming* (*ReGeP*) is a small java package which was developed in the context of this work to tackle this problem and allow the function set to be effectively decoupled from the application or framework in which it is used. The *Java package for evolutionary art* (*Jpea*) is a second package, also developed in the context of this work, which demonstrates the usage of *ReGeP* in an evolutionary art application.

Kurzfassung

Genetische Programmierung ermöglicht die automatische Lösung von Problemen durch die Vorgabe eines Funktionssatzes, aus dem Individuen basierend auf Prinzipien der natürlichen Evolution erzeugt, bewertet, selektiert, mutiert und rekombiniert werden. *Reflection* beschreibt die Fähigkeit eines Programms, seine eigene Struktur zu kennen (und ggf. zu verändern), im Falle objektorientierter Programmiersprachen insbesondere auch die Möglichkeit, Informationen über Klassen und Objekte zur Laufzeit auslesen zu können. Evolutionäre Kunst ist ein Teilgebiet der künstlichen Kunst, bei der Bilder z.B. unter Verwendung von Genetischer Programmierung erzeugt und evolviert werden. Das definierende Element eines GP-Systems ist sein Funktionssatz, denn er legt die Größe des Lösungsraums und die Art

der möglichen Lösungen fest. Unterschiedliche Frameworks fordern unterschiedliche Arten von Funktionssätzen, was eine Neuimplementierung der genutzten Funktionen für fast jede Applikation bedeutet. *Reflection-based Genetic Programming* (*ReGeP*) ist ein kleines Java-Paket, das im Rahmen dieser Arbeit entwickelt und implementiert wurde, das sich dieses Problems annimmt und eine Entkopplung des Funktionssatzes von der Applikation und dem Framework, in dem er verwendet wird, ermöglicht. Das *Java package for evolutionary art* (*Jpea*) ist ein zweites Paket, das entwickelt wurde, um die Funktionsweise von *ReGeP* anhand einer Applikation zur Generierung evolutionärer Kunst zu demonstrieren.

Kapitel 1

Einleitung

Genetische Programmierung (GP), ein Teilgebiet der Evolutionären Algorithmen (EA), ist eine Technik zur automatischen Lösung von Problemen durch den Computer unter Nutzung evolutionärer Prinzipien wie Selektion, Mutation und Rekombination. Im Allgemeinen wird dafür ein Funktionssatz vorgegeben, aus dem ein Genotyp variabler Länge erzeugt wird. Dies ist der große Unterschied zu den Genetischen Algorithmen (GA), bei denen wesentlich mehr über den Genotyp bekannt sein muss und er eine konstante Größe hat (Poli, Langdon und McPhee 2008; Meffert 2012).

Reflection bezeichnet eine Technik, durch die einem Programm Informationen über die Komponenten der Laufzeitumgebung (wie z.B. Klassen) zur Verfügung gestellt werden. Beim *Reflection-based Genetic Programming* soll der Funktionssatz völlig vom eigentlichen GP-Prozess entkoppelt werden, um eine höhere Wiederverwendbarkeit und weniger Anpassungsaufwand beim Einbinden neuer GP-Funktionen zu ermöglichen (Lucas 2004).

Evolutionäre Kunst bezeichnet eine Form der Kunst, bei der Bilder generiert und nach evolutionären Prinzipien weiterentwickelt werden. Die ersten Bilder werden dabei zufällig erzeugt, weitere Bilder durch evolutionäre Operatoren wie Mutation und Kreuzung und unter Selektion durch eine Bewertungsfunktion generiert.

Viel der Literatur zu evolutionärer Kunst stammt schon aus den 1980er-oder 1990er-Jahren, wo diese Art der computergenerierten Kunst durch größere Rechenleistungen erstmals wirklich interessant wurde. Karl Sims war einer der Pioniere auf

diesem Gebiet; er erzeugte *LISP*-Ausdrücke, die als Genotyp dienten und aus denen wiederum Bilder generiert wurden (Sims 1991). Seitdem wurden dieser und ähnliche Ansätze wie beschrieben in (Lewis 2008) intensiv untersucht. Eine ähnliche Art von Genetischer Programmierung soll auch in dieser Projektarbeit Verwendung finden, denn eine der einfachsten Arten, Bilder von annehmbarer Komplexität und Ästhetik zu schaffen, sind komplexwertige Funktionen, aus deren Rückgabewert die Farbwerte einzelner Pixel abgeleitet werden. Damit ähnelt dieses Projekt *jene*[1], das auf Sims' Ansätzen basiert, ist aber wesentlich modularer und erweiterbarer gestaltet und baut auf einem existierenden Framework auf.

Ein schwieriges Problem der Evolutionären Kunst war immer schon die Bewertungsfunktion, denn Kunst lässt sich noch nicht zufriedenstellend maschinell bewerten. Oft wird hier ein interaktiver Ansatz gewählt, d.h. der Nutzer entscheidet über die Ästhetik der erzeugten Bilder. Dieser Ansatz erlaubt allerdings nur eine sehr langsame Evolution und es ist unrealistisch, eine hohe Generationsanzahl zu erreichen. Eine Alternative ist die Nachbildung existierender Bilder, bei der die Bewertung evolvierter Bilder von der Ähnlichkeit zum Zielbild abhängt. Beide Ansätze sollen in dieser Arbeit verfolgt werden.

1.1 Zielsetzung

Ziel dieser Projektarbeit ist es, eine Software zu beschreiben, zu entwickeln und zu dokumentieren, die ähnlich wie in (Sims 1991) beschrieben mit Hilfe von Genetischer Programmierung komplexwertige mathematische Funktionen generiert, deren Rückgabewert den Farbwert des Pixels an der durch die Eingabeparameter identifizierten Position bestimmt. Dadurch soll insgesamt ein Bild generiert werden, das dann durch Benutzerinteraktion oder automatisch bewertet wird, um durch diese Bewertung dann die weitere Bildevolution zu beeinflussen.

Das Projekt gliedert sich softwaretechnisch insgesamt in zwei Programme, die wiederum aus zwei Modulen bestehen:

[1] "A lightweight evolutionary art package for Java"; `https://code.google.com/p/jene/`, besucht am 27.03.2013.

- *Reflection-based Genetic Programming* (*ReGeP*): Dieses Paket ist ein kleines Framework, das Genetische Programmierung über einen durch *Reflection* aufgebauten GP-Funktionssatz erlaubt.
  - `regep-core`: Das Kernmodul funktioniert unabhängig von dem verwendeten GP-Framework und baut lediglich eine Datenbank des zu verwendenden GP-Funktionssatzes auf.
  - `regep-jgap`: Auf dem Kernmodul aufsetzendes Modul, das *JGAP* als Framework benutzt und den GP-Funktionssatz in eine *JGAP*-kompatible Form bringt.
- *Java package for evolutionary art* (*Jpea*): Dieses Programm baut auf *ReGeP* auf und beinhaltet u.a. die eigentliche Darstellung der Bilder und die Bewertungsfunktion.
  - `jpea-core`: Kernmodul, das frameworkunabhängige Logik und den strukturellen Rahmen des Pakets enthält.
  - `jpea-jgap`: Auf dem Kernmodul aufsetzendes Modul, das *JGAP* als Framework benutzt und verschiedene Arten von Bildgenerierung und -bewertung unterstützt.

1.2 Gliederung

Diese Arbeit gliedert sich in 6 Kapitel.

Kapitel 1 besteht neben dieser Gliederung aus einer einleitenden Übersicht über das zu bearbeitende Thema sowie aus einer Zielsetzung, in der die zu entwickelnde Software und Dokumentation beschrieben wird.

In Kapitel 2 wird anschließend auf die verwendeten Frameworks und Bibliotheken eingegangen, warum sie ausgewählt wurden, welchen Zweck sie erfüllen und wie sie in das Projekt eingebunden sind.

Die Kapitel 3 und 4 enthalten die Dokumentation der beiden Programme *ReGeP* und *Jpea*. Diese besteht neben der Erläuterung der zugrundeliegenden Techniken

und Ideen aus der Beschreibung der jeweiligen groben Struktur, der einzelnen Entwurfsentscheidungen bis hin zu den einzelnen Komponenten und deren Funktionen.

Kapitel 5 enthält eine Zusammenfassung der Arbeit sowie Beispielergebnisse.

Kapitel 6 schließlich gibt einen Ausblick auf mögliche Weiterentwicklungen der Arbeit, insbesondere auch im Hinblick auf die Bachelorthesis, die auf dieser Arbeit aufbauen soll.

Kapitel 2

Frameworks und Bibliotheken

2.1 Evaluation der Frameworks

Im Folgenden sollen die vier Frameworks *ECJ*, *GenPro*, *JGAP* und *Watchmaker* auf die Nutzbarkeit im Kontext dieses Projekts hin untersucht werden.

Da GP zur Evolution bildgenerierender Funktionen genutzt werden soll ist die Unterstützung von GP ein wichtiges Kriterium bei der Auswahl des Frameworks. Weiterhin soll sich die Funktionsbibliothek möglichst leicht und ohne größere Anpassungen oder Erweiterungen ändern lassen, somit ist eine Konfiguration des Evolutionsprozesses zur Laufzeit ein weiteres wichtiges Kriterium. Schließlich soll noch in Betracht gezogen werden, wie gut und umfassend das Framework dokumentiert ist, wie schwierig also eine Einarbeitung wäre und wie es um die Entwicklung und Wartung des Frameworks steht, um abschätzen zu können, wie zukunftstauglich ein Framework ist.

Folgende Evaluationskriterien lassen sich also bilden:

- Unterstützung von GP
- Möglichkeit zur vollständigen Konfiguration zur Laufzeit
- Zu erwartende Einarbeitungszeit / Dokumentation
- Wartung und Weiterentwicklung des Frameworks

2.1.1 ECJ

> "ECJ is a research EC system written in Java. It was designed to be highly flexible, with nearly all classes (and all of their settings) dynamically determined at runtime by a user-provided parameter file."[1]

Java Evolutionary Computation Toolkit (*ECJ*) ist ein umfangreiches Framework, das neben *JGAP* eins der bekanntesten Frameworks für Evolutionäre Programmierung ist.

Es wurde begründet von Sean Luke, Professor an der George Mason University, und wurde und wird von ihm und einer Vielzahl von Mitwirkenden entwickelt.

Zu den allgemeinen Features zählen ein GUI, das u.a. statistische Informationen anzeigt, Parameter-Dateien, die eine Laufzeitkonfiguration erlauben sowie Multithreading.

EC-Features umfassen z.B. über das Netzwerk verteilte Inselmodelle, verschiedene Evolutionsstrategien, viele verschiedene Selektionsoperatoren, Subpopulationen und -spezies und das Auslesen von Populationen aus Dateien.

GP wird durch Bäume repräsentiert und unterstützt u.a. *ephermeral random constants* (Poli, Langdon und McPhee 2008, p. 20), *automatically defined functions* (Koza 1994), Wälder mit mehreren GP-Bäumen, sechs verschiedene Baum-Erstellungs-Algorithmen und acht Beispielimplementierungen.

Zwar erlauben die Parameter-Dateien eine Konfiguration zur Laufzeit, die GP-Funktionen müssen jedoch in Form von Klassen angegeben werden. Da die GP-Funktionen dieses Projekts jedoch dynamisch generiert werden sollen und somit durch Objekte abgebildet werden, müssten hierfür zur Laufzeit Klassen erzeugt werden, um *ECJ* nutzen zu können.

Da *ECJ* schon seit über einem Jahrzehnt existiert (eine genaue Historie lässt sich nicht finden, das älteste herunterladbare Release hat den Zeitstempel April 2000) und bis heute weiterentwickelt wird (letzter Commit des SVN-Repository datiert auf November 2012) ist es sicherlich eines der am besten getesteten und dokumentierten offenen EC-Frameworks am Markt, auch wenn moderne Java-

[1] `http://cs.gmu.edu/~eclab/projects/ecj/`, besucht am 27.03.2013

Features (wie z.B. *Generics*) noch nicht einbezogen wurden.

Genetische Programmierung	GP wird unterstützt, auch erweiterte Konzepte wie *ephermeral random constants* und *automatically defined functions* sind implementiert.
Laufzeitkonfiguration	Zwar ist eine Konfiguration zur Laufzeit durch Parameter-Dateien möglich, diese setzen jedoch wie bereits oben beschrieben klassenbasierte GP-Funktionen voraus.
Einarbeitungszeit	*ECJ* ist sehr komplex, aber bereits lange auf dem Markt und hat eine entsprechend ausführliche und umfangreiche Dokumentation, die eine Einarbeitung erleichtert.
Wartung und Weiterentwicklung	Wird kontinuierlich weiterentwickelt, auch wenn das letzte Release von Ende 2010 ist und das nächste Release laut Ankündigung auf der *ECJ*-Webseite nicht vollständig abwärtskompatibel sein soll.

2.1.2 GenPro

> "The aim of GenPro is to add Genetic Programming (GP) to the developer's general toolbox. This opposed to where GP mostly is exercised: at academic levels with long learning curves, complex programming and cumbersome extension work."[2]

GenPro ist ein kleines Framework, das speziell für den Einsatz im Bereich Genetische Programmierung entwickelt wurde und auch *Reflection*-basierte Genetische Programmierung vollständig implementiert. Es unterstützt die Generierung von Java-Code und ist ohne großen Konfigurations-oder Erweiterungsaufwand nutzbar.

[2] `https://code.google.com/p/genpro/`, besucht am 27.03.2013.

Leider existiert praktisch überhaupt keine Dokumentation für *GenPro*, einzig Beispiele und JavaDoc, die jedoch eine Einarbeitung nicht wesentlich erleichtern. Der Code ist kaum dokumentiert, was die Einarbeitung weiter erschwert.

Es werden – soweit aus einer oberflächlichen Analyse des Codes ersichtlich – nur einfache GP-Features unterstützt (z.B. Basisoperatoren zur Mutation und Kreuzung), es ist also schwer zu beurteilen, ob nicht hier noch wesentliche Erweiterungen selbst implementiert werden müssten.

Genetische Programmierung	Wird in Grundzügen unterstützt, keine erweiterten Features.
Laufzeitkonfiguration	Wird über *Reflection* unterstützt.
Einarbeitungszeit	Trotz des eher geringen Umfangs wäre die Einarbeitung aufgrund fast gänzlich fehlender Dokumentation erheblich schwieriger als bei den anderen untersuchten Frameworks.
Wartung und Weiterentwicklung	Wird nicht mehr aktiv weiterentwickelt, letztes Release vom Dezember 2009.

2.1.3 JGAP

> "JGAP (pronounced "jay-gap") is a Genetic Algorithms and Genetic Programming component provided as a Java framework. It provides basic genetic mechanisms that can be easily used to apply evolutionary principles to problem solutions."[3]

JGAP ist das zweite große Framework im Bereich *Evolutionary Programming* neben *ECJ*, das diesem im Funktionsumfang jedoch leicht unterlegen ist.

Hauptentwickler und Autor der meisten Dokumentation auf der *JGAP*-Webseite ist Klaus Meffert, das Projekt wird jedoch ähnlich wie *ECJ* von einer Gruppe von

[3] `http://jgap.sourceforge.net/`, besucht am 27.03.2013.

Entwicklern und Mitwirkenden weiterentwickelt. Das Framework existiert seit etwa 8 Jahren.

Ziel des Frameworks ist u.a. große Modularität und einfache Erweiterbarkeit, Hauptaugenmerk wird außerdem auf die Dokumentation sowie die Qualität und Stabilität des Codes gelegt. Die Dokumentation ist umfassend, umfangreich und detailliert und erlaubt eine einfache Einarbeitung.

Der Funktionsumfang ist, vor allem auch was GP betrifft, grob mit *ECJ* vergleichbar (verschiedene Baum-Erstellungsverfahren, mehrere Bäume parallel etc.).

Zwar ist die Laufzeitkonfiguration nicht direkt möglich, diese lässt sich jedoch einfach implementieren, da GP-Funktionen über Objekte (und nicht Klassen) identifiziert werden und diese zur Laufzeit bestimmt werden können.

Genetische Programmierung	Wird in ähnlichem Maße wie von *ECJ* unterstützt.
Laufzeitkonfiguration	Ist zwar nicht eingebaut, lässt sich aber problemlos implementieren.
Einarbeitungszeit	*JGAP* hat eine umfangreiche und ausführliche Dokumentation, die die Einarbeitung erleichtern.
Wartung und Weiterentwicklung	Wird noch aktiv weiterentwickelt, das letzte Release liegt etwas mehr als ein halbes Jahr zurück.

2.1.4 Watchmaker

> "The Watchmaker Framework is an extensible, high-performance, object-oriented framework for implementing platform-independent evolutionary/genetic algorithms in Java."[4]

Watchmaker ist ein relativ junges und modernes Framework, das jedoch besonders im Bereich Genetische Programmierung noch wenige Features bietet.

[4] `http://watchmaker.uncommons.org/`, besucht am 27.03.2013.

Wichtige Features sind sonst ähnlich wie bei *ECJ* und *JGAP* z.B. *Multi-Threading*, Inselmodelle, verschiedene Evolutionsstrategien, darüber hinaus ist *Watchmaker* wenig invasiv, d.h. die zu entwickelnden Objekte sind vom Framework entkoppelt, außerdem wird die Möglichkeit zur benutzergesteurte Evolution bereits mitgeliefert, auch wiederverwendbare Swing-Komponenten für eigene GUIs werden angeboten.

Genetische Programmierung wird im Moment überhaupt nicht unterstützt, es wird jedoch eine Beispielanwendung mitgeliefert, die zeigt, wie man GP integrieren könnte, der Entwicklungsaufwand wäre natürlich an der Stelle wesentlich höher als bei den anderen Frameworks.

GP ist zwar für das nächste Release geplant, der letzte Commit des SVN-Repository liegt jedoch bereits über 1 Jahr zurück, es ist also in naher Zukunft nicht mit einem Release zu rechnen.

Genetische Programmierung	Wird noch nicht unterstützt, ist aber beispielhaft bereits umgesetzt und müsste entsprechend auf dieses Projekt übertragen werden.
Laufzeitkonfiguration	Da GP selbst implementiert werden müsste ließe sich auch die Laufzeitkonfiguration problemlos umsetzen.
Einarbeitungszeit	Die Dokumentation ist umfangreich, da jedoch Genetische Programmierung bisher noch nicht eingebaut ist und selbst implementiert werden müsste, wäre eine tiefere Einarbeitung in das Framework nötig.
Wartung und Weiterentwicklung	Wird noch weiterentwickelt, das letzte Release liegt allerdings schon über zwei Jahre zurück, der letzte SVN-Commit ein Jahr.

2.1.5 Fazit

ECJ und *JGAP* sind die verbreitetsten Frameworks und halten ihre Versprechen von Funktionsumfang, Stabilität und Dokumentation weitestgehend. *GenPro* und *Watchmaker* sind eher Nischenprodukte, die (noch) nicht alle Funktionen und / oder die nötige Dokumentation bieten, um einen schnellen und problemlosen Einsatz zu gewährleisten.

Watchmaker disqualifiziert sich im Prinzip nur durch die nicht vorhandene GP-Unterstützung, alle anderen Features, insbesondere die Hilfsmittel zur GUI-Entwicklung sowie die eingebaute Möglichkeit zur benutzergesteuerten Evolution heben es von den anderen Frameworks ab. Sollte sich hier in Zukunft also noch etwas tun, wäre ein Umstieg auf *Watchmaker* ggf. in Betracht zu ziehen, für dieses Projekt soll es jedoch zunächst nicht verwendet werden.

GenPro scheint insgesamt eher ein unfertiger Prototyp mit vielen guten Ideen zu sein als ein etabliertes und stabiles Framework, das als Basis für dieses Projekt dienen könnte. Die Idee des *Reflection*-basierten Aufbaus des GP-Funktionssatzes soll in diesem Projekt aufgegriffen und selbst implementiert werden.

ECJ unterscheidet sich von *JGAP* primär durch die Möglichkeit der Laufzeitkonfiguration, die zwar einerseits Teil der Evaluationskriterien war, andererseits aber auch der ausschlaggebende Faktor ist, der gegen *ECJ* spricht. Die Parameter-Dateien von *ECJ*, die die komplette Konfiguration des Evolutionsvorgangs inklusive des kompletten GP-Funktionssatzes beinhaltet, erfordert das Angeben von Klassen für die einzelnen GP-Funktionen. Da *Reflection* genutzt werden soll, um den GP-Funktionssatz aufzubauen (und dieser Satz eben nicht nur aus Klassen, sondern z.B. auch aus Methoden bestehen kann), wäre eine Generierung von Klassen zur Laufzeit nötig, was zwar möglich, aber erheblich aufwändiger ist als die bei *JGAP* mögliche Vorgehensweise.

Die Entscheidung fiel somit auf *JGAP*, das modular genug ist, um eine Erweiterung und Anpassung an das Szenario zu erlauben, gut genug dokumentiert, um eine angemessen schnelle und tiefe Einarbeitung zu gewährleisten (auch der Code selbst ist dokumentiert) und auch alle wichtigen GP-/EC-Features bietet, die auch

ECJ auszeichnen.

Um aber auch in Zukunft das Umschwenken in die eine oder andere Richtung (*ECJ* oder *Watchmaker*, oder ein völlig anderes Framework) so einfach wie möglich zu gestalten, soll bei der Entwicklung darauf geachtet werden, wo immer möglich unabhängig vom verwendeten Framework zu programmieren und eine Abstraktionsschicht einzubauen, die ein einfaches Einsetzen eines andere Frameworks zum späteren Zeitpunkt ohne Änderungen der darüberliegenden Architektur ermöglicht.

2.2 JGAP

In diesem Abschnitt sollen die in *ReGeP* (Kapitel 3) und *Jpea* (Kapitel 4) verwendeten, sowie die für die Einbindung und Konfiguration wichtigen Elemente von *JGAP* beschrieben werden. Dazu wird zunächst in Abschnitt 2.2.1 Genetische Programmierung (GP) anhand einer kurzen Beschreibung allgemein eingeführt, anschließend wird in Abschnitt 2.2.2 untersucht, wie GP bei der Implementierung in *JGAP* abgebildet wurde. Der konkrete Ablauf der Evolution in *JGAP* wird in Abschnitt 2.2.4 vorgestellt, nachdem vorher die wichtigsten Konfigurationsmöglichkeiten von *JGAP* in Abschnitt 2.2.3 vorgestellt wurden.

2.2.1 Genetische Programmierung

Es gibt so viele Definitionen von GP, dass eine eigene Formulierung unnötig erscheint, darum hier der erste Satz aus dem Vorwort des Buchs, an dem sich die nachfolgende Beschreibung von GP orientieren soll:

> "Genetic programming (GP) is a collection of evolutionary computation techniques that allow computers to solve problems automatically."
> (Poli, Langdon und McPhee 2008)

Diese automatische Lösung von Problemen wird durch das Generieren und Evolvieren von Computerprogrammen realisiert. Diese Evolution wird durch ein Bewertungsmaß gesteuert und durch genetische Operationen wie z.B. Kreuzung (*crossover*) oder Mutation vorangetrieben.

GP kommt dabei nicht immer zu einer Lösung des Problems – daraus lässt sich jedoch keinesfalls schließen, dass keine Lösung existiert. Der Faktor Zufall sorgt dafür, dass GP immer wieder völlig neue und unerwartete Lösungsansätze hervorbringt und praktisch resistent gegenüber Sackgassen ist, in die herkömmliche Algorithmen zur Lösungsfindung oft laufen.

2.2.2 Struktur

GP-Programme (Interface `IGPProgram`) bestehen aus einer Reihe von Ästen (*branches*, Interface `IGPChromosome`), die für sich genommen wiederum Syntaxbäume (*syntax trees*) sind und aus Knoten (*nodes*, Klasse `CommandGene`) und Blättern (*leaves*, ebenfalls von der Klasse `CommandGene`) bestehen. Die Knoten sind Funktionen (*functions*), die einen Rückgabewert und eine unterschiedliche Anzahl von Parametern (*arity*) haben können; sie bilden zusammen mit den Terminalen (*terminals*, Klasse `Terminal`), die die Blätter bilden, den GP-Funktionssatz (*primitive set*). Bei einfachen GP-Problemen (Klasse `GPProblem`) existiert oft nur ein Syntaxbaum, dessen Wurzelknoten (*root node*) einen bestimmten Rückgabetyp haben muss.

2.2.3 Konfiguration

JGAP bietet mit der Klasse `GPConfiguration` eine umfassende Konfigurationsmöglichkeit der Evolution. Die wichtigsten Konfigurationsparameter sollen im Folgenden kurz beschrieben werden.

Fitnessfunktion

Über die Methode `setFitnessFunction(GPFitnessFunction)` kann die für die Bewertung der Individuen verwendete Fitnessfunktion gesetzt werden.

Fitnesswert-Interpretation

Über die Methode `setGPFitnessEvaluator(IGPFitness-Evaluator)` kann beeinflusst werden, wie die Fitnesswerte der Individuen

interpretiert werden (also z.B., ob niedrigere Fitnesswerte bevorzugt werden).

Selektion

Über die Methode `setSelectionMethod(INaturalGPSelector)` kann die verwendete Selektionsmethode gesetzt werden (Standard ist hier eine Tunierselektion).

Kreuzung

Über die Methoden `setCrossoverMethod(CrossMethod)`, `setCrossoverProb(float)`, `setMaxCrossoverDepth()` kann die zur Kreuzung verwendete Methode, die Kreuzungswahrscheinlichkeit sowie die maximale resultierende Tiefe eines gekreuzten Individuums beeinflusst werden. Zusätzlich kann über die Methode `setFunctionProb(float)` die Wahrscheinlichkeit konfiguriert werden, mit der eine Funktion als Kreuzungspunkt gewählt wird (mit der Restwahrscheinlichkeit wird ein Terminal gewählt).

Mutation

Über die Methode `setMutationProb(float)` kann die Wahrscheinlichkeit beeinflusst werden, mit der ein Knoten bei der Erstellung eines Programms mutiert wird.

Populationszusammensetzung

Über die Methode `setNewChromsPercent(double)` kann der prozentuale Anteil neu generierter Individuen an der Population in jeder Generation festgelegt werden.

Prototypen

Über die Methode `setPrototypeProgram(IGPProgram)` kann ein Prototyp für die Evolution festgelegt werden.

Individuenkomplexität

Die Methoden `setMaxInitDepth(int)` und `setMinInitDepth(int)` erlauben die Festlegung einer Ober-und Untergrenze für die Baum-

tiefe neu generierter Individuen. Weiterhin kann über die Methode `setStrictProgramCreation(boolean)` festgelegt werden, ob die Evolution abgebrochen werden soll, wenn bei der Erstellung eines Programms weder eine Funktion noch ein Terminal mit einem benötigten Rückgabetyp vorhanden ist; wenn dies auf `false` gesetzt wurde (Standardeinstellung), dann kann über die Methode `setProgramCreationMaxTries(int)` konfiguriert werden, wie oft im Falle einer solchen Situation erneut versucht werden soll, ein Individuum zu erstellen, bevor aufgegeben wird.

Individuenverifizierung

Über die Methode `setVerifyPrograms(boolean)` kann festgelegt werden, ob erstellte Individuen auf ihre Gültigkeit (d.h. ob Fitnesswerte für sie berechnet werden können) geprüft werden sollen.

Knotenvalidierung

Über die Methode `setNodeValidator(INodeValidator)` kann optional eine Knotenvalidierung konfiguriert werden, über die z.B. der Aufbau von generierten Individuen kontrolliert und beeinflusst werden kann.

2.2.4 Ablauf

Der erste Schritt bei GP ist die Initialisierung einer Ausgangspopulation. Hier wird zufällig eine Ausgangssituation für die weitere Evolution geschaffen. Es gibt verschiedene Verfahren zur Generierung der Anfangspopulation, wie z.B. *full*, *grow* und *ramped half-and-half* (Poli, Langdon und McPhee 2008, p. 13), die hier nicht näher behandelt werden sollen. Ziel dieser Verfahren ist es, nach bestimmten Prinzipien aus dem zugrunde liegenden *primitive set* eine bestimmte Anzahl von Individuen zu generieren, die als Lösungen des Problems in Frage kommen.

In *JGAP* muss dazu zunächst die Konfiguration (Klasse `GPConfiguration`) erzeugt werden, über die eine Vielzahl von Evolutionsparameter (wie z.B. Populationsgröße, Fitnessfunktion oder maximale initiale Baumtiefe) festgelegt werden können, die bereits im vorangegangenen Abschnitt 2.2.3 beschrieben wurden. Für die meisten dieser Parameter existieren Standard-Werte, die aber je

nach GP-Problem oft nicht sinnvoll sind. Danach müssen die Ein-und Ausgabetypen der einzelnen Äste (in *JGAP chromosome* genannt) festgelegt und der GP-Funktionssatz definiert werden. Die Erstellung der Ausgangspopulation kann dann über einen einzigen Methodenaufruf (der Methode `GPGenotype.randomInitialGenotype(...)`) gekapselt werden, bevor dann die Evolution über `GPGenotype.evolve()` gestartet werden kann.

Anschließend beginnt die Selektion. Dabei werden nicht einfach nur die besten Programme herausgefiltert, sondern Individuen mit einer besseren Bewertung (*fitness*) haben eine höhere Wahrscheinlichkeit, als Eltern neuer Kindprogramme ausgewählt zu werden, d.h. sie haben statistisch mehr Kinder als schlechter bewertete Programme. Es gibt auch hier wieder verschiedene Selektionsverfahren, eines der verbreitetsten ist die Turnierselektion (*tournament selection*), bei der jeweils eine bestimmte Anzahl von Individuen zufällig gewählt und dasjenige mit der besten Bewertung genommen wird – in diesem Fall werden also für jedes neue Individuum zwei Selektionsturniere durchgeführt und die Gewinner als Elternteile verwendet.

JGAP verwendet zur Bewertung die in der Konfiguration (Methode `GPConfiguration.setFitnessFunction()`) gesetzte Bewertungsfunktion, die die Klasse `GPFitnessFunction` erweitern muss, sowie zur Selektion der Elternprogramme den in der Konfiguration (Methode `GPConfiguration.setSelectionMethod()`) gesetzten Selektions-Algorithmus, der das Interface `INaturalGPSelector` implementieren muss.

Zur Generierung neuer Individuen werden wie bereits in Abschnitt 2.2.1 erwähnt genetische Operationen – z.B. Kreuzung (Klasse `CrossMethod`, ebenfalls in der Konfiguration setzbar) und Mutation (Interface `IMutateable`) – angewandt. Bei der Kreuzung werden Unterbäume (*subtrees*) der beiden Elternprogramme an bestimmten Kreuzungspunkten (*crossover points*) ausgetauscht.

Mutation geschieht in ähnlicher Weise, dabei wird an einem zufällig gewählten Mutationspunkt (*mutation point*) der Unterbaum durch einen neuen, zufällig generierten, ersetzt. Manchmal wird dies der Einfachheit halber als eine Kreuzung zwischen dem zu mutierenden Individuum und einem neuen, zufällig generierten Individuum implementiert. Eine weitere Art der Mutation ist die Punktmutation,

dabei werden einzelne Knoten ersetzt, anstelle ganzer Unterbäume.

Bei *JGAP* werden einzelne Knoten dadurch mutiert, dass die Methode `IMutateable.applyMutation()` mit einer bestimmten Prozentzahl aufgerufen wird. Eine höhere Prozentzahl ist Indikator für eine stärkere Mutation. Die Implementierung der Mutation bleibt dem Entwickler der jeweiligen `CommandGene`-Klasse überlassen. Das Objekt kann unverändert bleiben, es können kleinere Änderungen vorgenommen werden, aber es kann auch ein völlig neues Objekt generiert werden (was dann der oben beschriebenen Punktmutation entspricht).

Jede Evolution benötigt ein Terminierungskriterium, dies kann eine bestimmte Anzahl von zu evolvierenden Generationen sein, oder auch die Erreichung eines bestimmten Fitnessfunktions-Grenzwertes. Bei *JGAP* ist dieses Terminierungskriterium in der Regel die Generationsanzahl, die der Methode `GPGenotype.evolve()` übergeben wird.

2.2.5 Die `CommandGene`-Klasse

Ein `CommandGene`-Objekt stellt einen Knoten eines GP-Programms dar, die wichtigsten Eigenschaften und Methoden sollen hier kurz vorgestellt werden.

Stelligkeit und Rückgabewert

Die beiden definierenden Eigenschaften eines `CommandGene`-Objekts (und damit einer GP-Funktion) sind einerseits die Stelligkeit (*arity*) und andererseits ihr Rückgabetyp. Diese beiden Eigenschaften bestimmen, an welchen Stellen im GP-Programm die Funktion eingesetzt werden kann.

Die Stelligkeit legt die Anzahl der Eingabeparameter einer GP-Funktion fest. `CommandGene`-Objekte mit einer Stelligkeit von 0 sind Terminale – sie erwarten keine Eingabeparameter. Bei *JGAP* ist eine variable Stelligkeit implementiert, d.h. verschiedene Individuen der gleichen `CommandGene`-Klasse können eine unterschiedliche Stelligkeit haben. Zugriff auf die erwarteten Typen der einzelnen Kindknoten ist über die Methode `getChildType(IGPProgram individual, int childIndex)` möglich.

Der Rückgabetyp bestimmt den Typ des Knotens und damit, für welche anderen Knoten er als Kindknoten in Frage kommt. *JGAP* unterstützt, im Gegensatz zu einigen anderen GP-Frameworks, keinen mehrwertigen Rückgabewert. Möchte man also eine GP-Funktion mit unterschiedlichen Rückgabetypen erstellen, so muss man mehrere Instanzen der gleichen GP-Funktion erstellen, die dann jeweils einen eigenen Rückgabetyp haben können.

Entsprechend erwartet der minimale Konstruktor der `CommandGene`-Klasse lediglich eine Instanz der `GPConfiguration`-Klasse, die Stelligkeit und den Rückgabetyp der abgebildeten GP-Funktion.

Ausführung

Die Ausführung erfolgt über eine von sieben `execute_{returnType}(...)`-Methoden, zusätzlich gibt es noch eine typlose `execute(...)`-Methode, die selbst anhand des Rückgabetyps der GP-Funktion bestimmt, welche der anderen Methoden aufzurufen ist.

Hier eine Liste der standardmäßig definierten `execute_{returnType}(...)`-Methoden:

- `execute_boolean(...)`
- `execute_int(...)`
- `execute_long(...)`
- `execute_float(...)`
- `execute_double(...)`
- `execute_void(...)`
- `execute_object(...)`

Der zweite Teil des jeweiligen Methodennamens gibt den Rückgabetyp der Methode an. Da Java allerdings das automatische *Boxing* und *Unboxing* von primitiven Typen unterstützt (d.h. die Umwandlung in Objekte), würde die letzte Methode in

der Praxis völlig ausreichen, da der jeweils ausführende Elternknoten den erwarteten Rückgabetyp kennt.

Als Parameter erwarten alle `execute*(...)`-Methoden ein `ProgramChromosome`-Objekt, einen den Index des aktuellen Knoten repräsentierenden *Integer* sowie einen *Array* mit Parametern für die Ausführung.

Ein `ProgramChromosome` ist ein Ast des GP-Programms (repräsentiert durch die Klasse `GPProgram`). D.h. ein `GPProgram` besteht aus mehreren `ProgramChromosome`-Objekten, diese wiederum bestehen aus mehreren `CommandGene`-Objekten.

Das `CommandGene`-Objekt braucht bei der Ausführung Wissen über den Ausführungskontext (also sowohl das zugehörige `ProgramChromosome`, als auch den Index, an dem es sich darin befindet), da es selbst die Ausführung der eigenen Kindknoten (über die es jedoch keine Kenntnis besitzt) anstossen muss. Dies geschieht über die `execute*(...)`-Methoden der `ProgramChromosome`-Klasse, die anstelle des `ProgramChromosome`-Parameters einen *Integer* erwarten, der den auszuführenden Kindknoten identifiziert.

Listing 2.1 zeigt die Implementierung der `execute_int(...)`-Methode der *JGAP*-Klasse `Add` und verdeutlicht u.a., wie die Kindknoten ausgeführt werden.

Listing 2.1: `execute_int(...)`-Methode der `Add`-Klasse

```
public int execute_int(ProgramChromosome c, int n, Object[] args
    ) {
  return c.execute_int(n, 0, args) + c.execute_int(n, 1, args);
}
```

Abbildung 2.1 zeigt grafisch die Ausführung eines GP-Programms, das aus einer Addition von zwei Terminalen besteht (also aus drei Knoten), die hier kurz erklärt werden soll. Das `GPProgram`-Objekt ruft, nachdem die Ausführung von außen (z.B. bei der Auswertung des Individuums) über die `execute_int(int chromosomeNumber, Object[] args)`-Methode angestossen wurde, die `execute_int(Object[] args)`-Methode des Chromosoms mit der

übergebenen Nummer auf. Das `ProgramChromosome`-Objekt ruft dann wiederum die `execute_int(ProgramChromosome c, int nodeOffset, Object[] args)`-Methode des ersten Knotens (in diesem Fall ein Objekt der `Add`-Klasse) auf, als Parameter werden `this` (als Ausführungskontext, damit die Kindknoten ausgeführt werden können), `0` (es handelt sicht um den ersten Knoten des Baums, der intern als *Array* repräsentiert wird) und die bereits von außen übergebenen Argumente übergeben.

Nun wird der Code aus *Listing* 2.1 ausgeführt, es wird also auf dem übergebenen `ProgramChromosome`-Objekt die Methode `execute_int(nodeOffset, int child, Object[] args)` für beide Kindknoten aufgerufen. Der `nodeOffset` ist dabei in beiden Fällen `0` (da sich die Ausführung jeweils auf den aktuellen `Add`-Knoten bezieht, der den *Offset* `0` hat), während für `child` jeweils `0` und danach `1` übergeben wird, wodurch dann beide Kindknoten ausgeführt werden.

In der Methode `execute_int(int nodeOffset, int child, Object[] args)` der Klasse `ProgramChromosome` wird dann wiederum für jeden Kindknoten die Methode `execute_int(ProgramChromosome c, int nodeOffset, Object[] args)` ausgeführt, als `nodeOffset` wird nun jeweils die Position des Kindknotens im Gesamtbaum übergeben – sodass diese auch ggf. wieder ihre Kindknoten ausführen können. In unserem Beispiel handelt es sich bei den Kindknoten jedoch um `Terminal`-Objekte, die Blätter des Baums darstellen und keine weiteren Kindknoten haben, sie geben einfach den intern gespeicherten (oder einen generierten) Wert zurück.

Hat die Ausführung also die Blätter des Baums erreicht, werden die Ergebnisse sukzessive immer weiter hoch gereicht und auf jeder Ebene die den Knoten zugeordneten Funktionen mit den Ergebnissen aus den tieferen Ebenen ausgeführt, bis die Wurzel des Baums erreicht ist.

Das Feature, über die unterschiedlichen `execute`-Methoden eine Funktion für unterschiedliche Rückgabetypen zu definieren, wird im Folgenden nicht genutzt, sondern es wird in allen Fällen `execute_object(...)` verwendet (bei primitiven Typen werden entsprechend die *Wrapper*-Klassen verwendet). Gründe dafür

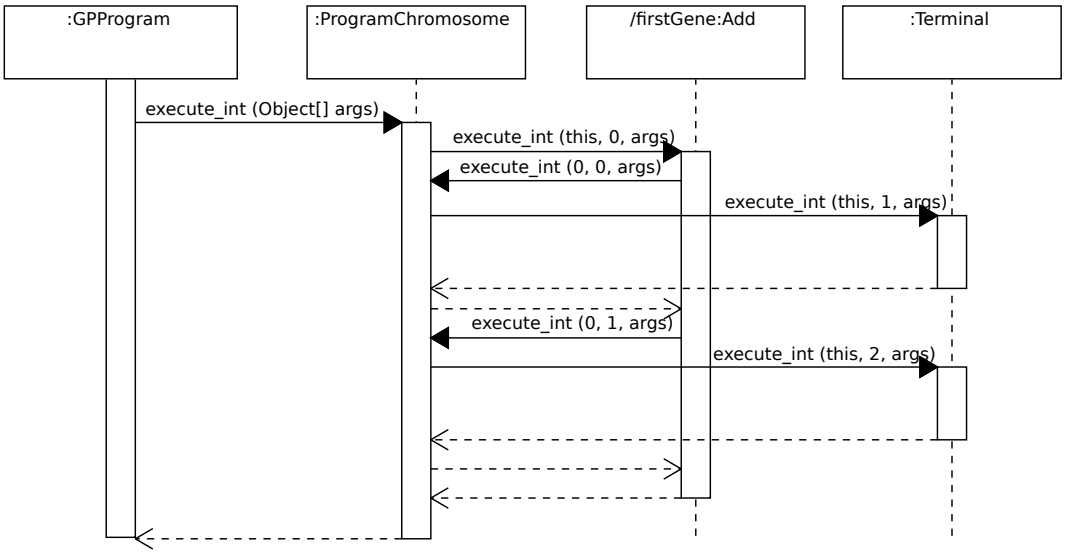

Abbildung 2.1: Beispielhafte vereinfachte Ausführungssequenz eines aus drei Knoten bestehenden GP-Programms.

werden in Abschnitt 3.6.2 ausgeführt.

2.3 Bibliotheken

Hier werden die verwendeten Bibliotheken kurz vorgestellt.

2.3.1 Reflections

> "Reflections scans your classpath, indexes the metadata, allows you to query it on runtime and may save and collect that information for many modules within your project." [5]

Die *Reflections*-Bibliothek erlaubt Zugriff auf Metadaten der Laufzeitumgebung (wie z.B. Klassen eines Pakets, Methoden und Felder von Klassen oder *Annotations*).

Verwendung findet diese Bibliothek in *ReGeP* zur automatischen Analyse der Laufzeitumgebung sowie u.A. zum Abfragen von Methoden von Klassen, die bestimmten Bedingungen entsprechend – allgemein dient die Bibliothek als Erweiterung des Java-eigenen `java.lang.reflect`-Pakets.

2.3.2 Guava

> "The Guava project contains several of Google's core libraries that we rely on in our Java-based projects: collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth."[6]

Die *Guava*-Bibliotheken bieten nützliche Erweiterungen der Java-Standardpakete. Eines der Einsatzgebiete bei *ReGeP* ist das explizite *Wrapping* von Klassen primitiver Datentypen (um Probleme mit den Unterschieden von z.B. `double.class` und `Double.class` zu umgehen).

[5] `https://code.google.com/p/reflections/`, besucht am 27.03.2013.

[6] `https://code.google.com/p/guava-libraries/`, besucht am 27.03.2013.

2.3.3 Marvin Image Processing Framework

> "Marvin is an extensible, cross-platform and open source image processing framework developed in Java."[7]

Das *Marvin Image Processing Framework* bietet neben Möglichkeiten zur Bildbe- und -verarbeitung wiederverwendbare GUI-Elemente und eine Plugin-Struktur, die eine problemlose Erweiterung des Funktionsumfangs erlaubt (Archanjo, Andrijauskas und Muñoz 2008).

Kern des Frameworks ist die `MarvinImage`-Klasse, die eine `BufferedImage`-Instanz kapselt und eine Vielzahl von Operationen zur Manipulation von Bildern (z.B. Zuschneiden und Skalieren) bietet.

Das Framework kommt in *Jpea* zum Einsatz und wurde ausgewählt, da es einfach zu verwenden ist, und für fast alle denkbaren Einsatzbereiche im Zusammenhang mit Bildern (Laden/Speichern, Manipulation, Anzeige im GUI) Funktionen bietet und durch *Plugins* problemlos noch zu erweitern ist.

Das *Marvin*-Framework kommt bei *Jpea* zur Verwaltung und Bearbeitung von Bildern zum Einsatz, es bietet mehr Funktionen als die von Java mitgelieferten Klassen und kapselt den Zugriff auf diese durch komfortablere Interfaces.

2.3.4 imgscalr

> "imgscalr is an very simple and efficient (hardware accelerated) 'best-practices' image-scaling library implemented in pure Java 2D;[...]"[8]

imgscalr ist eine kleine auf die Skalierung von Bildern spezialisierte Bibliothek, die in *Jpea* u.A. zur Generierung der verkleinerten Individuenbilder in der Populationsübersicht genutzt wird.

---

[7] `http://marvinproject.sourceforge.net/en/index.html`, besucht am 27.03.2013.

[8] `http://www.thebuzzmedia.com/software/imgscalr-java-image-scaling-library/`, besucht am 27.03.2013.

Kapitel 3

Reflection-based Genetic Programming (ReGeP)

In diesem Teil soll das Software-Paket *Reflection-based Genetic Programming* (*ReGeP*)[1] beschrieben und dokumentiert werden. Es ist als eine Erweiterung für EC-Frameworks konzipiert, die bereits GP unterstützen, aber immer noch eine starke Kopplung zwischen Framework und GP-Funktionssatz erzwingen. Diese Kopplung soll *ReGeP* weitestgehend auflösen, indem mittels *Reflection*, ähnlich wie in (Lucas 2004) beschrieben, der Funktionssatz automatisch zur Laufzeit bestimmt werden soll.

3.1 Einleitung

Die Idee zu diesem Softwarepaket entstand aus der Tatsache, dass bei *JGAP* (wie auch bei *ECJ* und den meisten anderen Frameworks, die GP unterstützen) alle Elemente des GP-Funktionssatzes einem bestimmten Interface entsprechen bzw. eine bestimmte Klasse erweitern müssen und somit einerseits der Funktionssatz bereits zur Kompilationszeit feststehen und andererseits für jede einzelne GP-Funktion Hand angelegt werden muss.

Dabei können fast alle Elemente von Klassen in Java ohne Probleme als GP-Funktionen betrachtet werden: Methoden haben einen Rückgabewert und keinen,

[1] `https://gitorious.org/regep`, besucht am 27.03.2013

einen oder mehrere Eingabewerte – genau wie eine GP-Funktion. Konstruktoren können als GP-Funktionen betrachtet werden, die eine Instanz der Klasse zurückgeben, von der sie definiert wurden. Klassenkonstanten können als Terminale betrachtet werden.

Das Ziel dieses Pakets ist also, über Reflection einen Rahmen und eine Verarbeitungsstruktur zu schaffen, über die automatisch Klassen in ihre Bestandteile zerlegt und diese Bestandteile wiederum in GP-Funktionen umgewandelt werden können.

Das folgende Zitat von (Lucas 2004) fasst *Reflection* in Objektorientierter Programmierung recht gut zusammen:

> "Reflection in an OO language is where a program is able to discover things about itself at run time, such as the class of an object, the fields associated with it, its superclass, the set of interfaces it implements, its set of constructors, and the set of methods that can be invoked on it." (Lucas 2004)

Lucas' Idee war es, objektorientierte Programme zu evolvieren. Also Programmcode zu erzeugen, der sich ausführen lässt, und nicht mehr Ausdrucksbäume, wie es bei GP üblich ist. Sein Ansatz ähnelt in der Umsetzung der linearen GP, da für eine Methodenimplementierung einfach eine Liste von Anweisungen erzeugt wird, die mutiert werden kann (durch Hinzufügen, Entfernen oder Ersetzen von Anweisungen).

ReGeP übernimmt hiervon primär die Idee, über *Reflection* die auf einem Objekt bzw. einer Klasse aufrufbaren Methoden auszulesen, um sie dann für GP nutzbar zu machen. Der so erzeugte GP-Funktionssatz lässt sich sowohl in einem dem von Lucas beschriebenen ähnlichen Szenario, als auch in klassischer Ausdrucksbaum-basierter GP nutzen.

Das Paket gliedert sich grob in zwei Teile, einerseits in das Modul `regep-core`, das frameworkunabhängig ist und einen Großteil der Umwandlung übernimmt, und andererseits in frameworkspezifische Module – zunächst nur `regep-jgap` –, die für den letzten Schritt der Umwandlung in für das jeweilige Framework passende GP-Funktionen zuständig sind.

3.2 Struktur

Einzelne GP-Funktionen, wie sie in Abschnitt 3.3 beschrieben sind, werden über das Interface `regep.core.gp.function.Function` bzw. über dieses Interface implementierende Klassen abgebildet.

Der Kern von `regep-core` wird durch eine Verarbeitungsstruktur gebildet, die auf dem Interface `regep.core.processing.Processor` aufbaut, diese wird in den Abschnitten 3.4 und 3.5 erläutert. Ziel dieser Verarbeitung ist es, Klassenelemente wie Methoden und Konstruktoren in GP-Funktionen zu überführen (d.h. in Klassen zu kapseln, die das Interface `Function` implementieren).

3.3 Funktionen

Funktionen (Interface `Function`) bilden GP-Funktionen und -Terminale ab, und werden definiert über eine beliebige Anzahl von Eingabe-Parametern (hat eine Funktion keine Eingabeparameter, so handelt es sich um ein Terminal) und einen Rückgabewert. Über die Methode `Function.execute(Object[] args)` kann eine GP-Funktion ausgeführt werden.

Das Interface `CachingFunction` stellt eine Funktion dar, über die die Rückgabewerte einer Funktionausführung zwischengespeichert werden können. Somit muss die Funktion nicht erneut ausgeführt werden, wenn sich an den Eingabeparametern nichts geändert hat. Zunächst wurde dies allerdings nur für Terminale, also Funktionen ohne Eingabeparameter, aktiviert, da das Prüfen der Eingabeparameter in vielen Fällen (zumindest im Rahmen der hier genutzen GP-Funktionen) länger dauert als die Ausführung der Funktion selbst. Näheres dazu in Abschnitt 3.3.1.

Die so abgebildeten Funktionen haben jedoch keine Kenntnisse von der Struktur des GP-Programms (also der Baumstruktur), diese muss vom GP-Framework umgesetzt werden. Einem GP-Knoten wird dann eine `Function`-Instanz zugewiesen, der dann die entsprechenden Eingabeparameter übergeben werden.

Fehler bei der Ausführung einer Funktion (z.B. aufgrund ungültiger Parameter/-Kombinationen) werden nicht behandelt, sondern an den Aufrufer weitergereicht,

der damit umgehen muss, siehe dazu auch Abschnitt 4.1.2.

Im Folgenden sollen die einzelnen konkreten `Function`-Klassen vorgestellt werden.

3.3.1 `CachingFunctionWrapper`

Diese Klasse implementiert das bereits erwähnte Interface `CachingFunction` und bildet einen *Wrapper*, der bei der `execute()`-Methode einen Cache zwischenschaltet, der nur bei geänderten Eingabeparametern die Funktion erneut ausführt, um dadurch die Performance bei der GP-Programmauswertung zu verbessern. Performance-Tests und Statistiken über *Caching-Hits* und *-Misses* ergaben jedoch, dass dies nur bei Terminalen, also Funktionen ohne Eingabeparametern, sinnvoll ist. Für alle anderen Funktionen ist der Overhead des Vergleichs der Eingabeparameter zu groß. Eine mögliche Verbesserung dieses Vorgehens wäre, die Äste des GP-Baums nach Variablen zu durchsuchen – denn nur die können sich ja ändern – und *Caching* für alle Äste zu aktivieren, in denen sich keine Variablen befinden. Interessanter wäre Caching auch wiederum, wenn der GP-Funktionssatz sehr komplexe Funktionen aufweist, deren erneute Ausführung den Mehraufwand für die *Caching*-Implementierung rechtfertigen würde. Da im vorliegenden GP-Funktionssatz jedoch primär relativ simple mathematische Funktionen vorkommen (einen groben Einblick in den GP-Funktionssatz gibt der Abschnitt 4.2), sollen diese Überlegungen hier zunächst nicht weiter verfolgt werden.

3.3.2 `ConstructorFunction`

Diese Funktion kapselt die Ausführung eines Konstruktors zur Erstellung eines Objekts. Die Eingabeparameter entsprechen denen des Konstruktors und der Rückgabewert ist das vom Konstruktor instanziierte Objekt.

3.3.3 `InstanceMethodFunction`

Diese Funktion kapselt die Ausführung einer Instanz-Methode. Für die Ausführung einer Instanzmethode wird natürlich ein Objekt der Klasse benötigt, die die Metho-

de deklariert hat. Somit ist der erste Eingabeparameter ein solches Objekt, während die restlichen Parameter sowie der Rückgabewert denen der Methode entsprechen.

3.3.4 `StaticMethodFunction`

Diese Funktion kapselt die Ausführung einer statischen Methode. Dies ist im Vergleich zur Instanz-Methode unproblematischer, da sich die Methode einfach direkt ohne eine Instanz der Klasse aufrufen lässt. Somit entsprechen sowohl Ein-, als auch Ausgabeparameter direkt denen der statischen Methode.

3.4 Processing

Die oben beschrieben Funktionen kapseln die Elemente einer Klasse in GP-kompatible Funktionen, der nächste Schritt besteht also darin, den eigentlichen Kapselungsprozess zu implementieren.

In Abbildung 3.1 ist die angestrebte Verarbeitungsreihenfolge abgebildet, mit der aus Paketnamen oder Klassen schließlich der GP-Funktionssatz erzeugt werden soll.

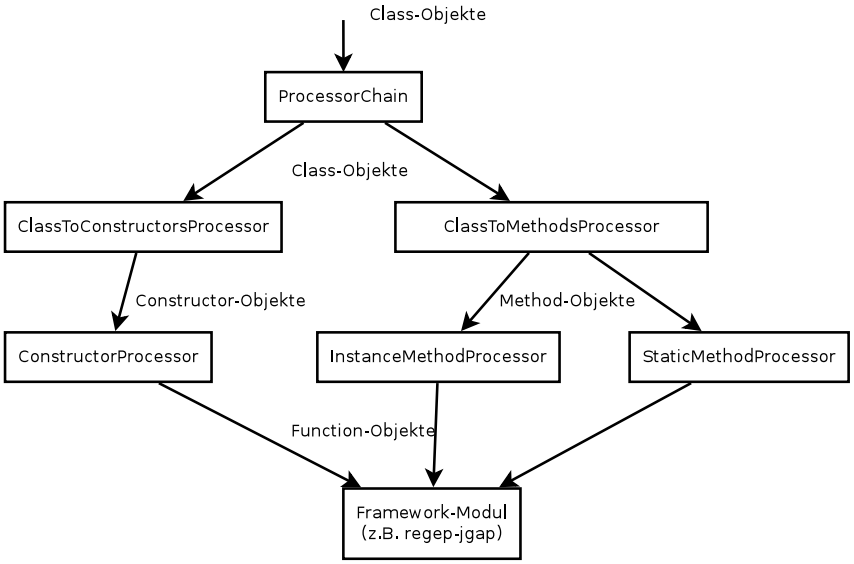

Abbildung 3.1: Verarbeitungshierarchie im Modul `regep-core`.

3.4.1 `Processor` und `ProcessorChain`

Die Idee der Verarbeitung besteht darin, dass es `Processor`-Klassen gibt, die aus einem Eingabeelement (welche Elemente ein Prozessor verarbeiten kann, wird über das `Applicable`-Interface kommuniziert) beliebig viele Ausgabeelemente erzeugen. Über eine `ProcessorChain` werden mehrere Prozessoren zu einer Verarbeitungseinheit verbunden. Diese Verarbeitungseinheit wird über den Aufruf der `process(Object)`-Methode des `ProcessorChain`-Objekts angestossen, wodurch das erste Eingabeelemente in allen Prozessoren verarbeitet wird (die Verarbeitung findet natürlich nur statt, wenn der Prozessor das jeweilige Objekt auch verarbeiten kann). Die `ProcessorChain` hat nun dafür zu sorgen, dass alle Ausgabeelemente wiederum in allen Prozessoren verarbeitet werden. Damit hat man eine stufenweise Verarbeitung erreicht, ohne den Mehraufwand der Modellierung einer tatsächlichen Hierarchie. Jeder Prozessor weiß, welche Elemente er verarbeiten kann und welche nicht, somit bleibt lediglich der Mehraufwand, dass jeder Prozessor für jedes zu verarbeitetende Element aufgerufen wird und prüfen muss, ob es verarbeitet werden kann. Eine notwendige Bedingung für das Funktionieren dieser Verarbeitung ist, dass die Methode `isApplicable(Object)` für jedes Ausgabeelement eines Prozessors `false` zurück geben muss – sonst ist kaum sicherzustellen, dass die Verarbeitung terminiert. In der Klasse `RecursiveFiltering-ProcessorChain` ist zusätzlich ein Schutz vor Endlosrekursion eingebaut, der abbricht, wenn Elemente der gleichen Klasse auf verschiedenen Rekursionsebenen verarbeitet werden. Abbildung 3.2 zeigt den prinzipiellen Ablauf der Verarbeitung.

Die genaue Implementierung der `ProcessorChain` ist nicht vorgegeben, die erste Implementierung durch die Klasse `RecursiveFilteringProcessor-Chain` verarbeitet die Elemente rekursiv und bietet zusätzlich über die Implementierung des Interfaces `FilterChain` die Möglichkeit zur Filterung aller zu verarbeitender Elemente.

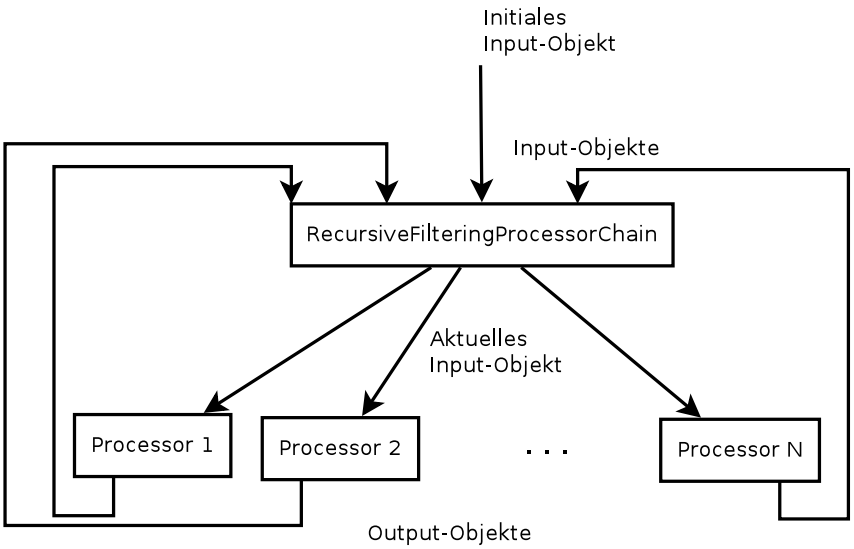

Abbildung 3.2: Allgemeiner Ablauf der Verarbeitung in der `RecursiveFilteringProcessorChain`.

3.5 Die `Processor`-Klassen

Hier sollen kurz alle konkreten `Processor`-Klassen vorgestellt und beschrieben werden, die bereits in Abbildung 3.1 eingeführt wurden.

- `PackageProcessor`: Dieser Prozessor interpretiert Zeichenketten (Eingabe) als Paketnamen und gibt, sofern das Paket existiert, eine Liste der darin enthaltenen Klassen zurück (Ausgabe).

- `ClassProcessor`: Diese abstrakte Oberklasse aller Klassen-Prozessoren implementiert nur die `isApplicable(Object input)`-Methode, die für `Class`-Objekte `true` zurück gibt.

- `ClassToMethodsProcessor`: Dieser Prozessor extrahiert aus `Class`-Objekten (Eingabe) eine Liste aller als `public` deklarierten Methoden (Ausgabe).

- `ClassToConstructorProcessor`: Dieser Prozessor extrahiert aus `Class`-Objekten (Eingabe) eine Liste aller `public` deklarierten Konstruktoren (Ausgabe).

- `StaticMethodProcessor`: Erstellt eine `StaticMethodFunction` (Ausgabe), die als *Wrapper* für die statische Methode (Eingabe) dient.

- `InstanceMethodProcessor`: Erstellt eine `InstanceMethod-Function` (Ausgabe), die als *Wrapper* für die Instanz-Methode (Eingabe) dient.

- `ConstructorProcessor`: Erstellt eine `ConstructorFunction` (Ausgabe), die als *Wrapper* für den Konstruktor (Eingabe) dient.

Somit wurden am Ende für alle Methoden und Konstruktoren der Klassen GP-Funktionen erstellt, die nun wiederum über einen frameworkspezifischen Prozessor weiterverarbeitet werden können. Mehr dazu in Abschnitt 3.6.2.

3.5.1 `FunctionDatabase`

Die `FunctionDatabase` ist eine Klasse, die von `regep-core` zur Verfügung gestellt wird, um Endprodukte der Verarbeitung zu speichern und nach außen (z.B. dem GP-Framework) ein Interface zum Abrufen des GP-Funktionssatzes zur Verfügung zu stellen.

Das Modul `regep-core` selbst verwendet die Datenbank nicht, da üblicherweise die letzten Verarbeitungsschritte von frameworspezifischen `Processor`-Klassen durchgeführt werden (in diesem Fall von `JgapFunctionAdapter-Processor`, siehe Abschnitt 3.6.2) und erst danach das Speichern in der Datenbank sinnvoll ist.

Die wichtigste Methode der Klasse ist `getFunctionSet()`, die einen validierten Funktionssatz zurück gibt. Bei der Validierung wird für jede Funktion geprüft, ob sie in einem GP-Programm verwendet werden könnte. Dazu wird zunächst geprüft, ob es eine andere Funktion gibt, die den Rückgabewert der zu prüfenden Funktion als Parameter erwartet (also als Elternknoten verwendet werden könnte) oder ob die Funktion als Wurzelknoten eines GP-Programms verwendet werden könnte. Ist diese Voraussetzung erfüllt, wird geprüft, ob es für jeden Parameter der

Funktion eine andere, gültige Funktion gibt, die einen entsprechenden Rückgabewert besitzt (also ob es mögliche Kindknoten für die Funktion gibt).

3.6 Das Modul `regep-jgap`

Dieses frameworkspezifische *ReGeP*-Modul dient dazu, die vorgestellten `Function`-Klassen mit einem weiteren Adapter zu umgeben, der sie für *JGAP* verwendbar macht. Abbildung 3.3 zeigt die Weiterverarbeitung der `Function`-Objekte aus dem `regep-core` Modul bis hin zur Übergabe des GP-Funktionssatzes an *JGAP*.

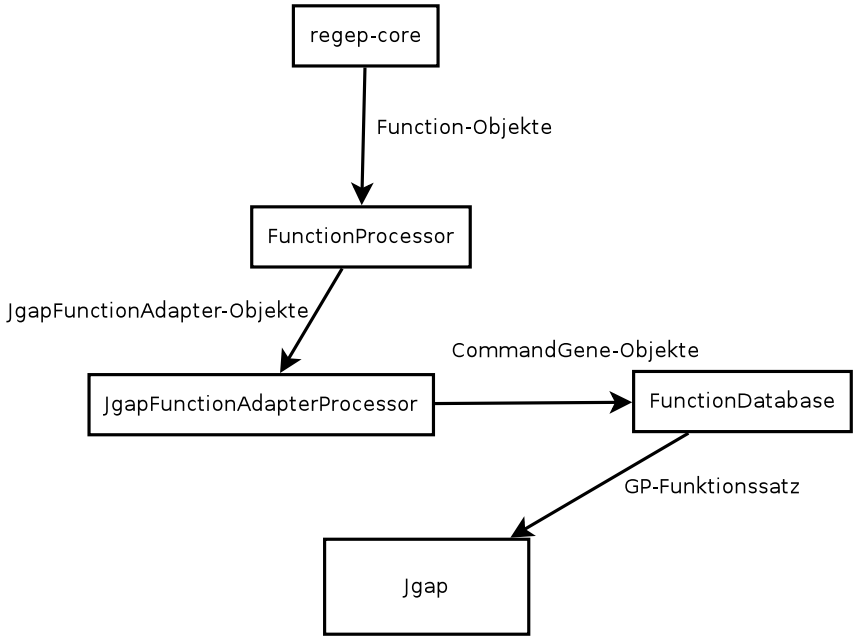

Abbildung 3.3: Prinzipielle Verarbeitungsstruktur des `regep-jgap`-Moduls.

3.6.1 Der Adapter: `JgapFunctionAdapter`

Diese Klasse erweitert die *JGAP*-Klasse `CommandGene` und implementiert sämtliche von dieser Klasse benötigten Methoden. Die meisten davon werden direkt oder indirekt an die unterliegende `Function`-Instanz weitergegeben.

Zusätzlich implementiert der `JgapFunctionAdapter` jedoch die strukturelle GP-Umsetzung, die in der `Function`-Klasse außen vor gelassen wird, denn jeder Knoten muss bei *JGAP* selbstständig für die Ausführung der Kindknoten sorgen. Dies ist in der privaten Methode `executeChildren()` gekapselt und wird beim Aufruf der `execute_{type}()`-Methoden automatisch mit ausgeführt.

Näheres zur `CommandGene` Klasse wurde bereits in Abschnitt 2.2.5 ausgeführt.

3.6.2 Die `Processor`-Klassen

Das `regep-jgap`-Modul stellt zwei `Processor`-Klassen bereit, mit denen der GP-Funktionssatz erstellt werden soll. Diese sollen im Folgenden kurz vorgestellt werden.

`FunctionProcessor`

Dieser Prozessor nimmt eine `Function`-Instanz (Eingabe) und erstellt einen `JgapFunctionWrapper` für sie. Es wird jedoch nicht nur ein *Wrapper* erzeugt, sondern, um größere Kombinationsmöglichkeiten in dem resultierenden GP-Funktionssatz zu erreichen, für jede Klasse in der Klassenhierarchie des Rückgabewerts ein eigener. Für eine Funktion mit dem Rückgabetyp `java.lang.Double` würden also beispielsweise drei *Wrapper* erzeugt: Einer mit Rückgabetyp `Double`, einer mit Rückgabetyp `Number` und einer mit Rückgabetyp `Object`. Somit ließe sich die gleiche Funktion als Kindfunktion für alle drei Typen verwenden. *JGAP* bietet zwar über die verschiedenen in Abschnitt 2.2.5 vorgestellten `execute`-Methoden eine begrenzte Möglichkeit, eine Funktion für verschiedene Rückgabetypen zu definieren, jedoch ist dies nur für primitive Typen vorgesehen und daher im Vergleich zu dieser Lösung unflexibel.

`JgapFunctionAdapterProcessor`

Dieser Prozessor verarbeitet die vom `FunctionProcessor` erzeugten `JgapFunctionAdapter` (die ja auch vom Typ `CommandGene` sind), er erzeugt je-

doch keine Ausgabe. Er speichert stattdessen alle verarbeiteten `CommandGene`-Objekte in einer und macht sie nach außen verfügbar. Somit stellt dieser Prozessor praktisch das Ende der Verarbeitung dar und ist auch der einzige, auf den außerhalb eine Referenz existieren sollte, um den GP-Funktionssatz abrufen zu können.

An dieser Stelle wird die bereits in Abschnitt 3.5.1 beschriebene Klasse `FunctionDatabase` verwendet, um die verarbeiteten Funktionen zu verwalten. Abbildung 3.4 zeigt den Aufbau der aus der Verarbeitung resultierenden Objekte.

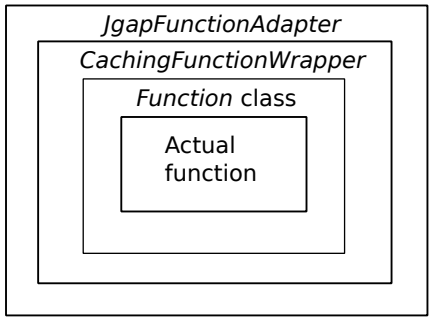

Abbildung 3.4: Schachtelungsstruktur der resultierenden GP-Funktionen.

Kapitel 4

Java package for evolutionary art (Jpea)

Das *Java package for evolutionary art* (*Jpea*)[1] ist ein kleines Softwarepaket, das einen Werkzeugkasten und eine Sammlung von Beispielapplikationen zu Evolutionärer Kunst darstellt. Es stellt einige grundlegende Mechanismen und Strukturen bereit, die bei der Entwicklung von EA-Applikationen wiederverwendet werden können und auf denen aufgebaut werden kann.

Ähnlich wie bei *ReGeP* gibt es eine frameworkunabhängige Schicht und Module für die einzelnen Frameworks (im Moment nur für *JGAP*). Es existiert keine prinzipielle Abhängigkeit zwischen *Jpea* und *ReGeP*, auch wenn die Beispielapplikation(en) *ReGeP* verwenden.

Zur Bildverarbeitung und -verwaltung wird das *Marvin Image Processing Framework* eingesetzt, das bereits in Abschnitt 2.3.3 kurz eingeführt wurde sowie *imgscalr* (siehe Abschnitt 2.3.4) zur Bildskalierung.

4.1 Das Kernmodul `jpea-core`

Wie in Abbildung 4.1 zu sehen ist, sind die beiden Hauptbestandteile des Kernmoduls von *Jpea* einerseits ein Interface zur Bilderstellung (`Phenotype-Creator`, siehe Abschnitt 4.1.2) und andererseits ein Interface zur Bildbewer-

[1] `https://gitorious.org/jpea/`, besucht am 27.03.2013

tung (`Evaluator`, siehe Abschnitt 4.1.3). Daneben kapseln die beiden Klassen `Population` und `Individual` (Abschnitt 4.1.1) die frameworkunabhängige Verwaltung der zu evolvierenden Individuen sowie deren Phänotypen und Bewertungen.

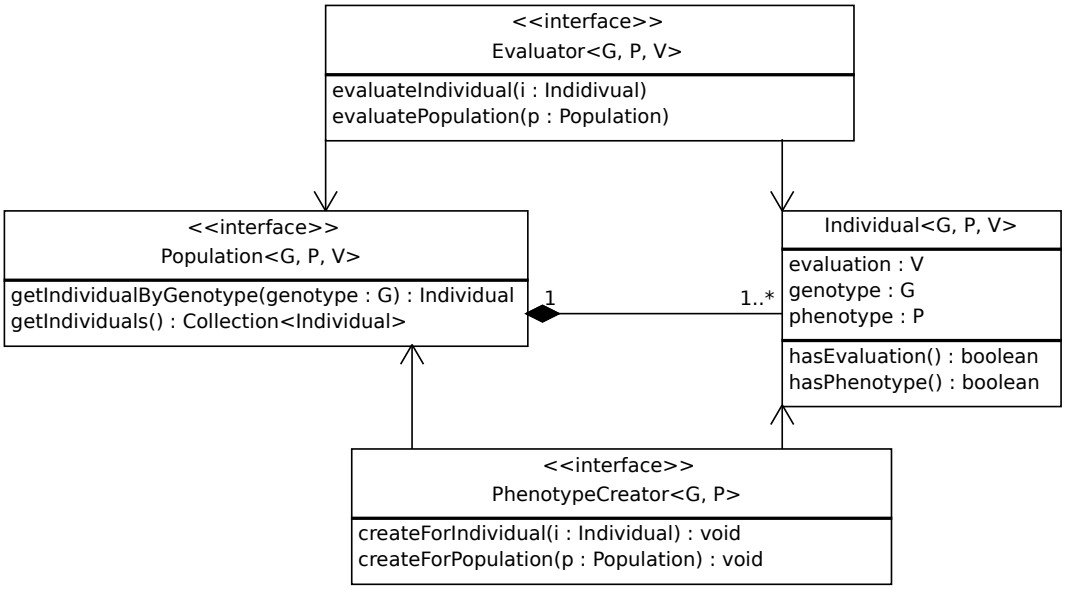

Abbildung 4.1: Übersicht der wichtigsten Elemente von `jpea-core`.

4.1.1 `Population` und `Individual`

Diese beiden Klassen bilden die Populationen der zu evolvierenden Individuen ab und kapseln deren Genotyp, Phänotyp und Bewertung. Um die Klassen möglichst flexibel zu gestalten, ohne Typinformationen zu verlieren, werden hier Java-*Generics* für alle drei Typen (also Genotyp, Phänotyp und Bewertung) eingesetzt.

Die Klasse `Individual` ist als *data transfer object* modelliert, dient also nur dem Speichern der genannten Daten und enthält keine weitere Logik. Sowohl bei der Phänotyperstellung (Methode `PhenotypeCreator.createForIndividual(Individual)`), als auch bei der Bewertung der Individuen (Methode `Evaluator.evaluateIndividual(Individual)`) werden die Daten direkt in den `Individual`-Objekten gespeichert und es wird nicht mit Rückgabewerten gearbeitet.

Individuen müssen zu jedem Zeitpunkt einen Genotyp haben, dieser wird dem Konstruktor übergeben und ist danach nicht mehr veränderbar. Zunächst hat ein Individuum also weder einen Phänotyp noch eine Bewertung, die beiden Methoden `hasEvaluation()` und `hasPhenotype()` müssen also vor dem direkten Zugriff auf Phänotyp oder Bewertung genutzt werden, wenn das Auftreten von `null`-Werten ausgeschlossen werden soll.

Individuen können auch nachhaltig ohne Phänotyp sein, im biologischen Sinne also nicht lebensfähig (d.h. die Erstellung des Phänotyps aus dem Genotyp ist fehlgeschlagen). Auf die technischen Gründe hierfür wird in Abschnitt 4.1.2 und auf die Auswirkungen in der Praxis in Abschnitt 4.2.3 näher eingegangen.

4.1.2 Bilderstellung – `PhenotypeCreator`

Durch GP wird zunächst ein Programm erzeugt, aus dem dann der `PhenotypeCreator` wiederum ein Bild (oder anderen Phänotyp) erstellen muss. Dies geschieht über die Methode `createForIndividual(Individual)`, der das Individuum übergeben wird, dessen Phänotyp erstellt werden soll.

Generics werden genutzt, um zwei der drei Typ-Parameter von `Individual` festzulegen (nämlich Genotyp und Phänotyp), wodurch Inkompatibilität zwischen `PhenotypeCreator` und dem übergebenen Individuum ausgeschlossen werden können. Der dritte Typ, die Bewertung, ist für den `PhenotypeCreator` irrelevant, da er mit der Bewertung nichts zu tun hat.

Als Ergebnis der Phänotyperstellung wird der erstellte Phänotyp im `Individual`-Objekt gespeichert. Es gibt Szenarien, in denen eine Phänotyperstellung nicht möglich ist, d.h. ein Fehler in der Ausführung des Algorithmus zur Phänotyperstellung aufgetreten ist. Dies ist vor allem im Hinblick auf einen leicht veränderbaren und vorher nicht bekannten GP-Funktionssatz denkbar. Beispiele wären z.B. die Division durch 0 oder negative Argumente für die Wurzel-oder Modulo-Funktion.

Weiterhin erlaubt die Methode `createForPopulation(Population)` eine Erstellung der Phänotypen für alle Individuen der Population. Dies ermög-

licht z.B. eine Parallelisierung der Phänotyperstellung, die aber bisher noch nicht implementiert ist. Die Klasse `AbstractPhenotypeCreator` bietet für diese Methode eine Standardimplementierung, die für alle Individuen der Population die Methode `createForIndividual(Individual)` aufruft.

Abbildung 4.2 zeigt die Hierarchie der `PhenotypeCreator`-Interfaces und -Klassen, deren wichtigste Elemente im Folgenden kurz beschrieben werden sollen.

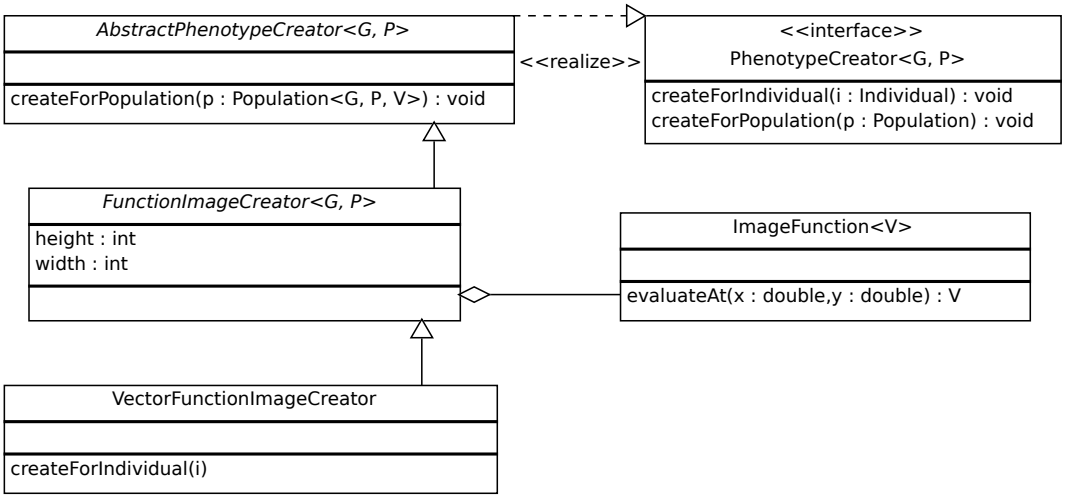

Abbildung 4.2: Klassendiagramm der `PhenotypeCreator`-Klassen und -Interfaces.

`FunctionImageCreator`

Der `FunctionImageCreator` erstellt allgemein Bilder aufgrund einer Funktion, die das Interface `ImageFunction` implementiert. Aus den Rückgabewerten der Funktion werden Farbwerte erzeugt, die Auswertungsstellen werden als Pixelkoordinaten interpretiert.

Die `ImageFunction` definiert nur eine einzige Methode, `evaluateAt(double x, double y)`, die die Bildfunktion an der gegebenen Stelle auswertet und ein Objekt eines durch *Generics* festgelegten Typs zurück liefert.

Eine konkrete Implementierung bietet die Klasse `VectorFunctionImageCreator`, die als Rückgabewert der `evaluateAt(double x, double y)`-Methode der verwendeten `ImageFunction` einen Vektor mit numerischen Kom-

ponenten erwartet. Dieser wird als RGB-Farbvektor interpretiert (d.h. die Komponenten entsprechen den Grundfarben) und auf diese Weise der Farbwert für jedes Pixel bestimmt und insgesamt ein Bild erzeugt.

4.1.3 Bildbewertung – `Evaluator`

Keine Evolution ohne Bewertungsfunktion: über dieses Interface wird die Methode `evaluateIndividual(Individual)` definiert, die eine Bewertung des übergebenen Individuums durchführt und diese dann darin speichert.

Diese Bewertung unterliegt prinzipiell keinen Einschränkungen. Je nach Implementierung können z.B. sowohl der Genotyp, als auch der Phänotyp zur Bewertung herangezogen werden, oder aber nur einer der beiden. *Generics* erlauben eine flexible Auswahl, welche Typinformationen zur Laufzeit zur Verfügung stehen müssen (z.B. kann eine Bewertungsfunktion, die nur den Phänotyp betrachtet, für beliebige Genotypen eingesetzt werden und umgekehrt).

Bewertungsfunktionen müssen aufgrund der in Abschnitt 4.1.2 beschriebenen Möglichkeit nicht existierender Phänotypen in der Lage sein, auch für solche Individuen eine Bewertung abzugeben, zumindest aber keinen Laufzeitfehler auszulösen. In den meisten Fällen dürfte es z.B. sinnvoll sein, Individuen ohne Phänotyp mit der schlechtesten Bewertung zu versehen.

Ähnlich wie beim `PhenotypeCreator` stellt auch das `Evaluator`-Interface eine Methode `evaluatePopulation(Population)` zur Verfügung, die z.B. eine Bewertung der Individuen in Abhängigkeit von anderen Individuen bzw. deren Bewertung erlaubt. Die Klasse `AbstractEvaluator` stellt eine Standardimplementierung dieser Methode zur Verfügung, in der die Methode `evaluateIndividual(Individual)` für alle Individuen der Population aufgerufen wird.

Abbildung 4.3 zeigt die wichtigsten Klassen und Interfaces, die mit dem `Evaluator`-Interface in Beziehung stehen und im Folgenden kurz beschrieben werden sollen.

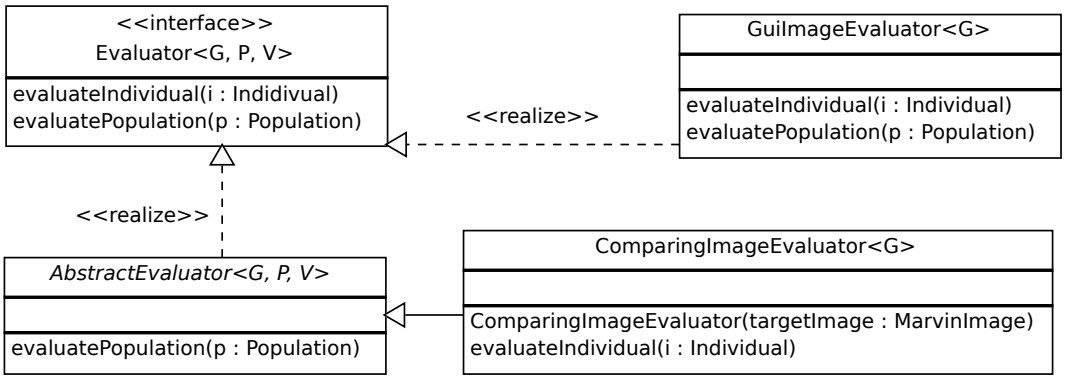

Abbildung 4.3: Klassendiagramm der `Evaluator`-Klassen und -Interfaces.

4.1.4 Applikationssammlung: `JpeaApplication`

Die Klasse `JpeaApplication` dient als Einstiegspunkt in eine Sammlung von verschiedenen Programmen, die im weiteren als Szenarios (Interface `Scenario` bezeichnet werden. Ein Szenario kapselt die beiden vorgestellten wichtigsten Elemente der Bildevolution (einen `PhenotypeCreator` und einen `Evaluator`), außerdem kann ein Konfigurator vorgeschaltet werden, der z.B. die Einstellung der Framework-Parameter erlaubt.

Zunächst wird nach dem Start eine Auswahl der vorhandenen Szenarios angezeigt, anschließend wird der jeweils verwendete Framework-Treiber sowie ggf. das Szenario selbst konfiguriert, bevor dann die Evolution gestartet wird.

Die konkreten Szenarios sind Teil des Moduls `jpea-jgap` und werden in den Abschnitten 4.2.3 und 4.2.4 vorgestellt.

4.2 Das Modul `jpea-jgap`

Das *JGAP*-Modul von *Jpea* bietet auf `jpea-core` aufbauend einige Klassen, die als Schnittstelle zu *JGAP* fungieren sowie einige konkrete Evolutionsszenarios.

Die Klasse `JgapDriver` (näher beschrieben in Abschnitt 4.2.1), die die Klasse `GPFitnessFunction` von *JGAP* erweitert, stellt eine *JGAP*-Bewertungsfunktion dar, die die komplette *JGAP*-Evolution kapselt. Sie überlässt die Bewertung einem von außen registrierten *Callback*-Objekt (Klasse `EvolutionEngine`, die

Bilderstellung und -bewertung kapselt).

Zur Bilderstellung dient die Klase `JgapImageCreator` (näher beschrieben in Abschnitt 4.2.2), die den `AbstractPhenotypeCreator` erweitert. Diese nutzt die Klasse `JgapImageFunction`, die das Interface `ImageFunction` implementiert, sowie einen `VectorFunctionImageCreator` zur Erstellung der Bilder.

Die Klasse `MathOperations` stellt einige einfache Funktionen (primär zur Generierung von Terminalen und zur Konvertierung zwischen verschiedenen Zahlen-Klassen) zur Erweiterung des GP-Funktionssatzes zur Verfügung.

Die Evolutionsszenarien sind schließlich in den Klassen `JgapInteractiveScenario` (Abschnitt 4.2.3) und `JgapComparingScenario` (Abschnitt 4.2.4) umgesetzt, die die Konfiguration von *JGAP* übernehmen und die Evolution anstoßen, nachdem über *ReGeP* der GP-Funktionssatz aus den Methoden der Klassen `MathOperations`, `java.lang.Math`, `com.google.common.math.IntMath`, `com.google.common.math.LongMath` und `com.google.common.math.DoubleMath` erstellt wurde. Als Frameworktreiber wird die Klasse `JgapDriver` verwendet, die durch die Klasse `JgapDriverConfigurator` konfiguriert wird und danach die komplette *JGAP*-Evolution übernimmt, über eine `EvolutionEngine` wird eine Bildbewertung und -erstellung damit verknüpft.

4.2.1 Der Frameworktreiber: `JgapDriver`

Diese Klasse sorgt dafür, dass *JGAP* ordnungsgemäß konfiguriert ist, erlaubt aber auch eine Änderung der *JGAP*-Konfiguration von außen. Sie erweitert die *JGAP*-Klasse `GPFitnessFunction` und registriert sich selbst bei *JGAP* als Bewertungsfunktion.

Wird die Methode `evaluate(IGPProgram)` dann aufgerufen, nutzt der Treiber die bei der Erstellung übergebene `EvolutionEngine`, die einen `PhenotypeCreator` und einen `Evaluator` kapselt und dafür sorgt, dass alle Individuen der aktuellen Generation bewertet werden, bevor die Evolution fortfährt.

Prinzipiell dient diese Klasse also als Brücke zwischen *JGAP*, das für jedes zu evaluierende Individuum die `evaluate(IGPProgram)`-Methode aufruft, und *Jpea*, das den Aufruf von `PhenotypeCreator.createForIndividual(Individual)` und `Evaluator.evaluateIndividual(Individual)` erwartet.

4.2.2 Bilderstellung

Für die Bilderstellung gibt es zunächst das Interface `PhenotypeCreator`, das bereits in Abschnitt 4.1.2 vorgestellt wurde, sowie den `FunctionImageCreator`, der auch bereits in Abschnitt 4.1.2 beschrieben wurde.

Das Modul `jpea-jgap` definiert auf dem `AbstractPhenotypeCreator` aufbauend die Klasse `JgapImageCreator`, deren primäre Aufgabe es ist, die *JGAP*-Programme vom Typ `IGPProgram` in einer `ImageFunction` zu verpacken und diese dann dem `VectorFunctionImageCreator` zur Auswertung zu überlassen, der dann (wie bereits in Abschnitt 4.1.2 beschrieben) daraus ein Bild erstellt.

Als `ImageFunction` wird die Klasse `JgapImageFunction` verwendet, deren `evaluateAt(double x, double y)`-Methode in der Lage ist, das verwaltete `IGPProgram`-Objekt auszuwerten, nachdem die darin enthaltenen Variablen auf die übergebenen Werte für `x` und `y` gesetzt wurden.

4.2.3 Interaktive Bildbewertung

Unter den Gesichtspunkten der eingangs dieser Arbeit genannten Problemen der automatischen und maschinellen Bildbewertung stellt die nutzergesteuerte Evolution, wie sie durch den `GuiImageEvaluator` realisiert wird, die naheliegendste Bewertungsfunktion dar.

Dieses Szenario nutzt zur Bildbewertung die bereits genannte Klasse `GuiImageEvaluator`, zur Bilderstellung den `JgapImageCreator` und als Frameworktreiber den `JgapDriver`.

Die Benutzeroberfläche bietet im oberen Bereich einen Überblick über alle Indi-

viduen der aktuellen Generation der Population. Unmittelbar neben den verkleinerten Bildern kann eine Bewertung in fünf Stufen – von "–" (sehr negativ) bis "++" (sehr positiv) – vorgenommen werden.

Um im Fall von Isomorphie – also Individuen mit unterschiedlichen Genotypen aber gleichen Phänotypen – eine mehrfache Bewertung der gleichen Bilder durch den Benutzer zu vermeiden, wird über die Klasse `PhenotypeDatabase` eine Datenbank von Phänotypen und zugehörigen Bewertungen aufgebaut. Durch diese wird vor der Bewertung durch den Benutzer geprüft, ob ein Individuum mit gleichem Phänotyp bereits zuvor evaluiert wurde und setzt ggf. die bereits vorhandene Bewertung, die dann vom Nutzer aber noch geändert werden kann.

Durch Anklicken der Bilder kann das Individuum zur Detailansicht ausgewählt werden. Diese findet im unteren Bereich des GUI statt und besteht im Moment aus einer Anzeige des Phänotyps in Originalgröße sowie einer Darstellung des Genotyps in Form einer Verzeichnisstruktur.

Die Anzeige des Genotyps ist folgendermaßen zu interpretieren: Der Wurzelknoten entspricht der Funktion `createColorVector()`, die drei Parameter erwartet, die eine Eben darunter angezeigt werden, wenn der Verzeichnisbaum expandiert wird. Die direkten Kindknoten entsprechen jeweils den Kindknoten der Funktionen im GP-Programm. Bei Terminalen (also den Blättern des Baums) wird zusätzlich der Wert des Terminals als String in eckigen Klammern hinter dem Funktionsnamen gelistet.

Durch den "Submit"-Knopf im unteren Bereich der Benutzeroberfläche kann die aktuelle Bewertung bestätigt werden und es wird mit der Evolution fortgefahren. Die Bewertungs-Knöpfe werden deaktiviert, bis die Phänotypen der neuen Individuen generiert wurden und die nächste Generation bewertet werden kann.

Da als Selektionsverfahren die Turnierselektion (siehe Abschnitt 2.2.4) angewendet wird, ist es möglich, dass einige Bilder der Population nicht ausgewählt werden und somit trotz möglicherweise guter Bewertung in der nächsten Generation nicht erneut auftauchen.

Im Laufe der Evolution kann es vorkommen, dass die konstante Populationsgröße (standardmäßig 10), zu schrumpfen scheint und im oberen Bereich nur noch

wenige Bilder angezeigt werden. Tatsächlich ist die Anzahl der Individuen weiterhin konstant, jedoch sind bei der Erzeugung der Bilder für die nicht angezeigten Individuen Fehler aufgetreten. Bei einer so geringen Populationsgröße und entsprechend geringer Diversität können sich solche Fehler natürlich rasch ausbreiten und damit die Anzahl der darstellbaren Individuen stark verringern.

Dies sollte aber nie zu langfristigen Probleme führen, da nicht darstellbare Individuen eine negative Bewertung erhalten und somit im weiteren Verlauf des Evoltionsprozesses rasch aussterben. Zudem besteht ein fester Anteil von Individuen jeder Generation aus völlig neu generierten Individuen, die der verringerten Diversität entgegenwirken.

Technische Gründe für nicht darstellbare Individuen wurden bereits in Abschnitt 4.1.2 erläutert.

4.2.4 Bewertung durch Bildvergleich

Dieses Szenario verwendet ebenfalls zur Bilderstellung den `JgapImageCreator` und als Frameworktreiber den `JgapDriver`, zusätzlich muss noch bei der Konfiguration das Zielbild ausgewählt werden. Zur Bildbewertung wird der `ComparingImageEvaluator` verwendet.

Der `ComparingImageEvaluator` realisiert eine Bildevolution, deren Ziel es ist, einem vorgegebenem Zielbild möglichst nahe zu kommen. Der Vergleichsalgorithmus ist dabei denkbar simpel (und entsprechend teilweise nicht zielführend): Der Abstand, den ein Bild vom Zielbild hat wird als Summe aller Pixelfarbwertdifferenzen angegeben, d.h. für jedes Pixel wird der Farbwert mit dem Farbwert des Zielbildes verglichen und ein numerischer Abstand dazu berechnet, dessen Aufsummierung über sämtliche Pixel des Bildes den Gesamtabstand zum Zielbild wiedergibt. Das Benutzerinterface zeigt einerseits das Zielbild an und andererseits das bisher beste Individuum, zusätzlich besteht die Möglichkeit, die Evolution zur pausieren (diese Pause tritt u.U. erst verzögert ein, da ein direkter Eingriff in den *JGAP*-Evolutionsprozess nicht möglich ist).

Kapitel 5

Zusammenfassung und Beispielergebnisse

In diesem Kapitel sollen zunächst die Ergebnisse und Erkenntnisse zusammengefasst werden, bevor anschließend einige Beispielergebnisse von *Jpea* vorgestellt werden.

5.1 ReGeP

Das Ziel bei der Entwicklung von *ReGeP* war es, eine einfache Erweiterung des GP-Funktionssatzes und eine Entkopplung von der Anwendung, in der er zum Einsatz kommt, zu erreichen. Dieses Ziel wurde durch eine automatische Umwandlung von Java-Klassenelementen in GP-kompatible Funktionen realisiert, die anschließend im Hinblick auf die Verwendbarkeit im vorhandenen GP-Problem hin gefiltert werden. Somit ist nach der Erstellung der `RecursiveFilteringProcessorChain` über einen Aufruf von

Listing 5.1: Verarbeitung einer Klasse durch die `ProcessorChain`

```
*/
```

die Übernahme aller Methoden der Klasse `java.lang.Math` in den GP-Funktionssatz möglich, ohne dass eine Anpassung an das verwendete Framework nötig ist. Diese Anpassung wird automatisch über frameworkspezifische Adapter-Klassen (im Fall des verwendeten *JGAP*-Frameworks ist dies die `JgapFunctionAdapter`-Klasse) umgesetzt.

Verwaltet wird der GP-Funktionssatz über die Klasse `FunctionDatabase`, die Kenntnis vom Wurzelknotentyp des zu erzeugenden GP-Programms hat. Ihre `getFunctionSet()`-Methode gibt entsprechend einen gefilterten GP-Funktionssatz zurück, der nur noch für das zu erzeugende GP-Programm verwendbare GP-Funktionen enthält.

Ermöglicht wird *ReGeP* im Grunde erst durch den Einsatz von *Reflection* zum Auslesen von Metadaten der Java-Klassen zur Laufzeit, was gleichzeitig eine Erweiterung und Veränderung des GP-Funktionssatzes ohne Code-Änderungen erlaubt.

Sämtliche Funktionsaufrufe der GP-Funktionen werden ebenfalls über die Java *Reflection*-API realisiert, wodurch sich bei der Auswertung der GP-Programme zur Bilderstellung (abhängig von der Anzahl Knoten des GP-Programms) Probleme ergeben können, da *Reflection*-Funktionsaufrufe ein Vielfaches der Zeit eines gewöhnlichen Funktionsaufrufes benötigen (Lucas 2004, p. 6).

Zwischenspeichern dieser Funktionsaufrufe wird zu diesem Zeitpunkt aus in Abschnitt 3.3.1 beschriebenen Gründen nur zur Realisierung von *random ephermeral constants* (Poli, Langdon und McPhee 2008, p. 20) verwendet.

5.2 Jpea

Jpea ist eine Toolbox rund um Evolutionärer Kunst, die *ReGeP* verwendet und neben einer Anwendung zur interaktiven Bildevolution auch die Nachbildung eines vorgegebenen Bildes durch Bildevolution beinhaltet.

Ziel war an dieser Stelle ein Einstieg in den Bereich der Evolutionären Kunst, der erst in der Bachelorthesis vertieft werden soll. Für diese wurden mit *ReGeP* und *Jpea* Grundlagen geschaffen, auf denen sich schnell und problemlos unterschiedli-

che Anwendungsszenarien der Evolutionären Kunst aufbauen lassen.

Der Fokus von *Jpea* liegt auf der Bilderstellung und -bewertung, die eigentliche Bildevolution – bzw. Evolution der die Bilder repräsentierenden Genotypen – wird hierbei in ein spezialisiertes Framework ausgelagert. Das erste und bisher einzige unterstützte GP-Framework ist *JGAP*.

Als Genotyp werden GP-Programme mit nur einem Syntaxbaum verwendet, deren Wurzelknoten einen Vektor zurück geben muss – in dem in den Beispielapplikationen verwendeten GP-Funktionssatz erfüllt diese Bedigung nur die Funktion `MathOperations.createColorVector(double, double, double)`, die ein Objekt der Klasse `Vector<Double>` zurück gibt, welches drei Komponenten hat. Weiterhin können zwei Variablen (`X` und `Y`) als Blätter im Syntaxbaum auftauchen.

Bei der Bilderstellung wird der zurück gegebene Vektor als RGB-Farbvektor interpretiert. Für jeden Pixel des zu erzeugenden Bildes werden die beiden Variablen auf die Koordinaten des Pixels gesetzt, bevor das GP-Programm ausgewertet wird.

5.3 Beispielergebnisse

Der GP-Funktionssatz umfasst neben einfachen mathematischen Operationen (Addition, Subtraktion, Multiplikation, Division, Modulo, Rundungsfunktionen etc.) primär trigonometrische Funktionen, daher sind Farbverläufe das vorherrschende Muster bei den evolvierten Bildern. Erzeugt werden Bilder mit einer Größe von 500x500 Pixel.

Im Folgenden sollen Beispielergebnisse anhand der erzeugten Bilder und den zugehörigen Genotypen vorgestellt werden.

Zur Notation der Genotypen: Auf jeder Zeile wird ein Knoten gelistet, die Einrückung gibt die Tiefe im Baum an, Kindknoten werden in den Zeilen jeweils nach den zugehörigen Elternknoten gelistet. Zur Erläuterung der einzelnen Funktionen sei auf die Dokumentation der Klassen `java-`

.lang.Math[1], com.google.common.math.IntMath[2], com.google-.common.math.LongMath[3] und com.google.common.math.Double-Math[4] verwiesen.

Die Entwicklung der generierten Bilder richtet sich grob nach eingestellter maximaler anfänglicher Baumtiefe (bei den generierten Beispielbildern war diese auf 4 gesetzt). Bei geringerer Baumtiefe (bis 4) herrschen in den ersten 20-30 Generationen Bilder mit ein-oder mehrfachen Farbverläufen oder einfarbige Bilder vor (siehe Abbildung 5.1). Danach werden die Bäume komplexer (die für die Evolution einzelner Generationen nötige Zeit erhöht sich) und es tauchen komplexere Strukturen in den Bildern auf (siehe Abbildungen 5.2 und 5.3). Eine höhere Komplexität der Genotypen bedeutet jedoch keineswegs automatisch auch komplexere Strukturen in den Bildern, hier sammeln sich oft Funktionen, die entweder den Funktionswert aus dem darstellbaren Bereich (0 bis 255) heraus tragen oder keine Auswirkungen auf das Ergebnis haben (z.B. das Runden bereits gerundeter Werte).

Die folgenden Abbildungen 5.1, 5.2 und 5.3 zeigen drei Beispielbilder und die Listings 5.2, 5.3 und 5.4 die zugehörigen Genotypen.

Listing 5.2: Genotyp zu Abbildung 5.1

```
createVolorVector()
␣␣factorial()
␣␣␣␣mod()
␣␣␣␣␣␣199
␣␣␣␣␣␣mod()
␣␣␣␣␣␣␣␣199
␣␣␣␣␣␣␣␣169
```

[1] http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html, besucht am 27.03.2013

[2] http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/math/IntMath.html, besucht am 27.03.2013

[3] http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/math/LongMath.html, besucht am 27.03.2013

[4] http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/math/DoubleMath.html, besucht am 27.03.2013

Abbildung 5.1: Beispielbild 1, Farbverläufe

```
␣␣hypot()
␣␣␣␣abs()
␣␣␣␣␣␣X
␣␣␣␣sub()
␣␣␣␣␣␣sub()
␣␣␣␣␣␣␣␣Y
␣␣␣␣␣␣␣␣X
␣␣␣␣␣␣X
␣␣hypot()
␣␣␣␣atan2()
␣␣␣␣␣␣102.555
␣␣␣␣␣␣196.298
␣␣␣␣sub()
␣␣␣␣␣␣Y
␣␣␣␣␣␣X
```

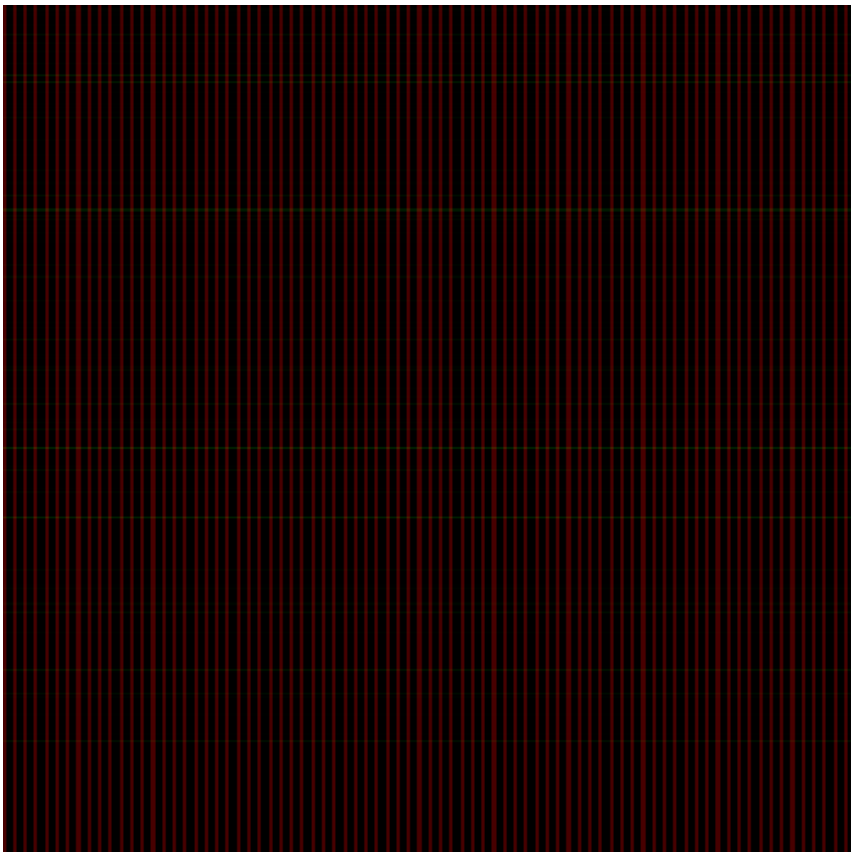

Abbildung 5.2: Beispielbild 2, Linienmuster

Listing 5.3: Genotyp zu Abbildung 5.2

```
createColorVector()
␣␣max()
␣␣␣␣checkedMultiply()
␣␣␣␣␣␣checkedSubtract()
␣␣␣␣␣␣␣␣190
␣␣␣␣␣␣␣␣119
␣␣␣␣␣␣round()
␣␣␣␣␣␣␣␣cos()
␣␣␣␣␣␣␣␣␣␣X
␣␣␣␣round()
␣␣␣␣␣␣cos()
␣␣␣␣␣␣␣␣X
␣␣nextAfter()
```

```
␣␣␣␣tan()
␣␣␣␣␣␣pow()
␣␣␣␣␣␣␣␣Y
␣␣␣␣␣␣␣␣116.383
␣␣␣␣cosh()
␣␣␣␣␣␣tan()
␣␣␣␣␣␣␣␣div
␣␣␣␣␣␣␣␣␣␣X
␣␣␣␣␣␣␣␣␣␣Y
␣␣log10()
␣␣␣␣tan()
␣␣␣␣␣␣pow()
␣␣␣␣␣␣␣␣Y
␣␣␣␣␣␣␣␣116.383
```

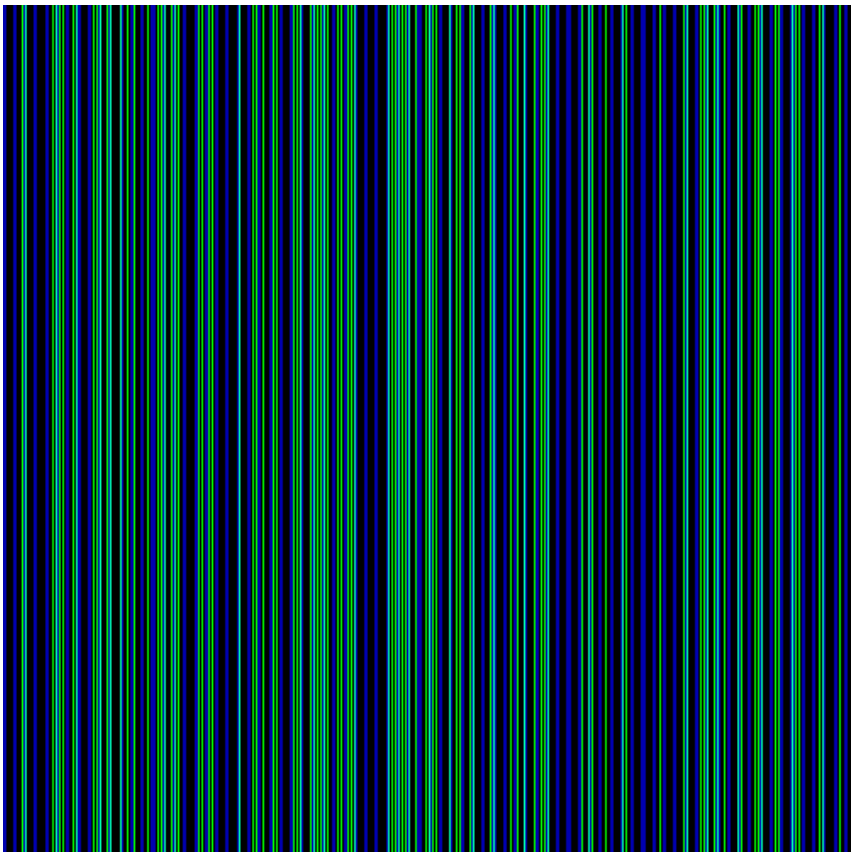

Abbildung 5.3: Beispielbild 3, unregelmäßige Muster

Listing 5.4: Genotyp zu Abbildung 5.3

```
createColorVector()
␣␣pow()
␣␣␣␣sin()
␣␣␣␣␣␣tan()
␣␣␣␣␣␣␣␣X
␣␣␣␣X
␣␣checkedAdd()
␣␣␣␣pow()
␣␣␣␣␣␣X
␣␣␣␣␣␣161
␣␣␣␣round()
␣␣␣␣␣␣cos()
␣␣␣␣␣␣␣␣X
␣␣max()
␣␣␣␣checkedMultiply()
␣␣␣␣␣␣checkedSubtract()
␣␣␣␣␣␣␣␣244
␣␣␣␣␣␣␣␣68
␣␣␣␣␣␣round()
␣␣␣␣␣␣␣␣cos()
␣␣␣␣␣␣␣␣␣␣X
␣␣␣␣round()
␣␣␣␣␣␣cos()
␣␣␣␣␣␣␣␣X
```

Kapitel 6

Ausblick

In diesem Kapitel sollen Themen und Aspekte kurz angerissen werden, die in dieser Arbeit nicht oder nicht erschöpfend behandelt wurden und damit ein Ausblick auf die Bachelorthesis gegeben werden, die auf dieser Arbeit aufbauen soll.

6.1 Reflection und GP

Bisher wurde im Zusammenhang mit *ReGeP* der GP-Funktionssatz einzig aus vorgegeben Klassen mit statischen Methoden aufgebaut. Hier könnte man *ReGeP* erweitern, sodass komplette `jar`-Archive geladen werden können und der GP-Funktionssatz aus allen darin enthaltenen Klassen aufgebaut wird.

Weiterhin ist es, insbesondere bei der Verwendung komplexerer GP-Funktionen, interessant, ein umfassenderes *Caching* zu implementieren, das den GP-Syntaxbaum analysiert und Äste ohne Variablen nur ein mal auswertet. Zur Verbesserung der Performance könnten Bibliotheken wie ReflectASM[1] zum Funktionsaufruf verwendet werden, die die deutlich langsamere (Lucas 2004, p. 6) Java Reflection-API ersetzt.

Auch weiterführende GP-Konzepte wie z.B. *automatically defined functions* (Koza 1994) könnten für eine Weiterentwicklung in Betracht gezogen werden.

Ein weiterer interessanter Punkt ist die persistente Speicherung von Genotypen, sodass sie in späteren Evolutionen wiederverwendet werden können. *JGAP* unter-

[1] `https://code.google.com/p/reflectasm/`, besucht am 27.03.2013

stützt zwar prinzipiell die Serialisierung von Genotypen, einerseits wäre allerdings eine frameworkunabhängige Umsetzung interessanter, andererseits ist bisher unklar, inwiefern die *JGAP*-Serialisierung mit den *ReGeP*-Funktionen kompatibel ist und eine Wiederherstellung ohne weiteres möglich wäre.

6.2 Evolutionäre Kunst

Im Zusammenhang dieser Arbeit wurde Evolutionäre Kunst nur oberflächlich eingeführt und verwendet, hierauf soll in der Bachelorthesis das Hauptaugenmerk liegen. Die verschiedenen Ansätze Evolutionärer Kunst, ihre Möglichkeiten sowie Vor-und Nachteile sollen intensiver untersucht werden.

Der GP-Funktionssatz könnte stark erweitert werden (z.B. um Fraktale), um eine größere Vielfalt bei den generierten Bildern zu erreichen und auch ästhetisch ansprechendere Bilder zu erzeugen. Im Hinblick auf die interaktive Bildevolution wäre eine Analyse und Anpassung der Funktionsergebnisse möglich, um möglichst viele Werte innerhalb des Wertebereichs (0 bis 255) zu erzeugen und damit komplett schwarze oder weiße Bilder oder konstante Werte einzelner Farbkomponenten zu vermeiden.

Eine wichtige Erweiterung für die Bilderzeugung und automatische Bewertung ist vor allem das *Multi-Threading*, wodurch beides stark beschleunigt werden könnte, auch die Evolution auf *JGAP*-Seite könnte durch Parallelisierung (die prinzipiell unterstützt wird) beschleunigt werden. In beiden Fällen ist eine Parallelisierung jedoch primär in nicht-interaktiven Szenarien interessant.

Neben den schon eingebauten Bewertungen durch den Benutzer und durch Bildvergleich wären maschinelle Bildbewertungen wie in (Heijer und Eiben 2010; Heijer und Eiben 2011; Ekárt, Sharma und Chalakov 2011) beschrieben oder eine interaktive Bewertung mit maschineller Unterstützung (also ein lernfähiger Bewertungsalgorithmus) interessant.

Literatur

Archanjo, G.A., F. Andrijauskas und D.R. Muñoz (2008). „Marvin–A Tool for Image Processing Algorithm Development“. In: *Technical Posters Proceedings of XXI Brazilian Symposium of Computer Graphics and Image Processing*, S. 5–6.

Ekárt, Anikó, Divya Sharma und Stayko Chalakov (2011). „Modelling human preference in evolutionary art“. In: *Proceedings of the 2011 international conference on Applications of evolutionary computation - Volume Part II*. EvoApplications'11. Torino, Italy: Springer-Verlag, S. 303–312. ISBN: 978-3-642-20519-4. URL: http://dl.acm.org/citation.cfm?id=2008445.2008480 (besucht am 27. 03. 2013).

Heijer, E. den und A. E. Eiben (2010). „Comparing aesthetic measures for evolutionary art“. In: *Proceedings of the 2010 international conference on Applications of Evolutionary Computation - Volume Part II*. EvoCOMNET'10. Istanbul, Turkey: Springer-Verlag, S. 311–320. ISBN: 3-642-12241-8, 978-3-642-12241-5. DOI: 10.1007/978-3-642-12242-2_32.

— (2011). „Evolving art using multiple aesthetic measures“. In: *Proceedings of the 2011 international conference on Applications of evolutionary computation - Volume Part II*. EvoApplications'11. Torino, Italy: Springer-Verlag, S. 234–243. ISBN: 978-3-642-20519-4. URL: http://dl.acm.org/citation.cfm?id=2008445.2008473 (besucht am 27. 03. 2013).

Koza, John R. (1994). *Genetic programming II: Automatic Discovery of Reusable Programs*. Complex adaptive systems. MIT Press, S. I–XX, 1–746. ISBN: 978-0-262-11189-8.

Lewis, Matthew (2008). *"Evolutionary Visual Art and Design", Kapitel aus The Art of Artificial Evolution*. Springer-Verlag.

Lucas, Simon M. (2004). „Exploiting Reflection in Object Oriented Genetic Programming". In: *European Conference on Genetic Programming*, to appear. URL: http://algoval.essex.ac.uk/rep/oogp/ReflectionBasedGP.pdf (besucht am 27. 03. 2013).

Meffert, Klaus (2012). *Genetic Programming with JGAP*. URL: http://jgap.sourceforge.net/doc/genetic_programming.html (besucht am 27. 03. 2013).

Poli, R., W. B. Langdon und F. McPhee (2008). *A Field Guide to Genetic Programming*. (With contributions by J. R. Koza). http://lulu.com. ISBN: 978-1-4092-0073-4. URL: http://www.gp-field-guide.org.uk/ (besucht am 27. 03. 2013).

Sims, Karl (Juli 1991). „Artificial evolution for computer graphics". In: *SIGGRAPH Comput. Graph.* 25.4, S. 319–328. ISSN: 0097-8930. DOI: 10.1145/127719.122752. URL: http://www.karlsims.com/papers/siggraph91.html (besucht am 27. 03. 2013).