GaFrame

Table of contents

1 GaFrame....2
1.1 Motivation....2
1.2 Genetische Algorithmen....2
1.3 Ein flexibles Framework....2
2 Konzeption....3
2.1 Anforderungen an ein Framework für Genetische Algorithmen....3
2.2 Einschränkungen....4
2.3 Entwurfsdetails zu GaFrame....4
3 Implementierung....10
3.1 Implementierte Verfahren....10
3.2 Parallelisierung....12
3.3 Verteilte Verarbeitung....13
4 Nutzung....15
4.1 Nutzung von GaFrame....15
4.2 Unterstützung für Multithreading....24
4.3 Verteilte Verarbeitung (Clusterbetrieb)....24
5 Downloads....26
5.1 Binärdistribution....26
5.2 Quelldistribution....26
5.3 Subversion-Repository....27
6 Entwicklerdokumentation....27
7 Export....27

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

1. GaFrame

1.1. Motivation

Dieses Projekt ist als Beleg im Fach *Genetische Algorithmen* (bei Prof. Iwe) an der HTW Dresden entstanden.

Projektmitglieder:

- Dominik Hölz (s52654)
- Wolfram Wagner (s52609)

1.2. Genetische Algorithmen

Unter dem Begriff *Genetische Algorithmen* sind Programme zusammengefasst, die versuchen, analytisch nicht oder nur unter enormem Aufwand zu berechnende Probleme zu lösen.

Ein Genetischer Algorithmus bedient sich dabei einer Vorgehensweise, die an das natürliche Vorbild, die Evolution, angelehnt ist:

- Jede mögliche Lösung des Algorithmus bildet ein *Individuum*
- Die Eigenschaften des gesuchten Programms (bzw. Algorithmus) werden in geeigneter Form (binär) codiert, sie bilden dann das *Chromosom* des Individuums
- Der Genetische Algorithmus bildet Mengen verschiedener Individuen(*Populationen*)
- Durch Kreuzung, Mutation oder weitere Operationen werden (analog zum natürlichen Vorbild) weitere Generationen erzeugt
- Eine *Fitnessfunktion* bestimmt die Qualität eines Individuums, legt also fest, wie gut dieses Individuum das Problem lösen kann
- Durch verschiedene *Auswahlverfahren* werden innerhalb mehrerer Generationen immer "fittere" Individuen erzeugt, bis eine vorgegebene Fitness erreicht ist

Ein einführender Artikel zu Genetischen Algorithmen, der einen kurzen Überblick über das Themenfeld gibt, findet sich bei Wikipedia.

1.3. Ein flexibles Framework

Da in zwei eigentlich unabhängigen Projekten (jeweils Belege im Fach Neuroinformationsverarbeitung) auch Genetische Algorithmen genutzt werden sollten, entstand die Idee, für diese beiden Projekte eine gemeinsame Basis zu schaffen.

Nach Diskussion der Anforderungen entstand so ein flexibel einsetzbares, auf Java

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

basierendes Framework, das bereits einige grundlegende Techniken der Genetischen Algorithmen zur Verfügung stellt, aber auch leicht an individuelle Bedürfnisse anpassbar ist.

Das Framework wird in den Belegen der Autoren (siehe oben) bereits erfolgreich eingesetzt, eignet sich aber auch für den Einsatz in weiteren Projekten.

2. Konzeption

2.1. Anforderungen an ein Framework für Genetische Algorithmen

Natürlich haben wir uns im Vorfeld gründlich Gedanken über die Fähigkeiten eines Frameworks für *Genetische Algorithmen (GA)* gemacht. Zu diesem Zweck wollen wir hier die von uns aufgestellten Anforderungen an ein solches System angeben und im Weiteren auf die konkrete Lösung eingehen.

Unsere Anforderungen waren:

1. allgemeine Anwendbarkeit
2. Grundstock an Verfahren
3. einfache Bedienung
4. Anpassbarkeit
5. performante Abarbeitung
    - Effizienz
    - Skalierbarkeit

Wie in der obenstehenden Listen zu sehen ist, müssen vielschichtige Problemstellungen beachtet werden. Die kurz aufgelisteten Punkte sollen jetzt noch ein wenig untersetzt werden:

allgemeine Anwendbarkeit:
Das System soll nicht auf spezielle Anwendungsfälle zugeschnitten sein, sondern sich auf möglichst viele Problemstellungen (die für natürlich für Genetische Algorithmen geeignet sein müssen) anwenden lassen. Hierzu ist es nötig, eine gute Chromosomen-Mimik zu wählen und Schnittstellen zu definieren, die es gestatten diese Mimik einfach nutzen zu können.
Grundstock an Verfahren:
In der Literatur sind zahlreiche Verfahren und Strategien beschrieben, mit denen man die einzelnen Phasen eines *GA* ausführen kann. Wir wollten einige nützliche Operatoren und Strategien zu Verfügung stellen, damit man gleich mit der Arbeit am Problem beginnen kann und nicht erst eigene Operatoren definieren muss.
einfache Bedienung:
Es soll einem Entwickler möglichst viel Arbeit erspart werden und kein großer Einarbeitungsaufwand nötig sein. Es handelt sich schließlich um ein kleines

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Framework und man sollte vermeiden, Sachverhalte zu verkomplizieren (gemäß dem Motto: keep it short and simple). Wenige Zeilen Code sollen genügen, um einen lauffähigen *GA* zu erzeugen. Weiterhin soll die Chromosomen-Mimik einfach nutzbar sein.

Anpassbarkeit:
Das System soll modular aufgebaut sein, so dass es möglich ist, Komponenten leicht auszutauschen oder anzupassen. Hierzu zählen zahlreiche Paradigmen des objektorientierten Designs (OOD). Es soll also möglich sein, eigene Komponenten (Klassen) unter Einhaltung der Schnittstellen in das System einzubringen, ohne das System an sich ändern zu müssen.

performante Abarbeitung:
Natürlich muss die Implementierung robust und gut entworfen sein. Geschwindigkeit war trotzdem unser oberestes Ziel. Weiterhin soll man auch von Mehrprozessormaschinen oder gar ganzen Rechnerfarmen profitieren und gemäß der Aufgabenstellung Möglichkeiten haben, den *GA* mit Rechenleistung zu versorgen.

2.2. Einschränkungen

Wir haben aus Gründen der einfachen Implementierung einige Einschränkungen festgelegt:

1. Chromosomen sind ausschließlich binär codiert (mittels `java.util.Bitset`).
2. Die Chromosomenlänge ist konstant.
3. Die Anzahl der Individuen pro Population ist konstant.

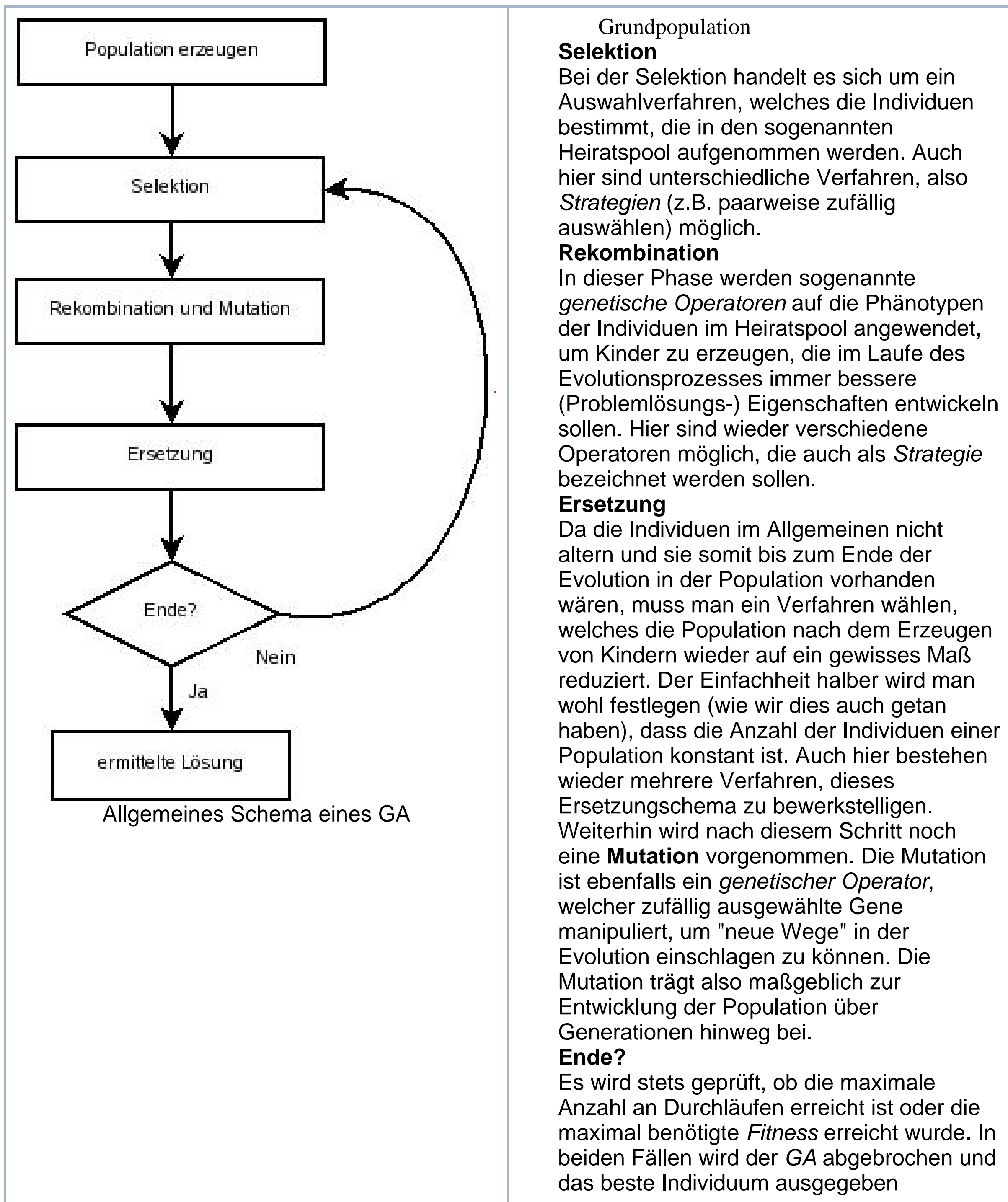
2.3. Entwurfsdetails zu GaFrame

2.3.1. Ablaufschema eines GA

Bewusst haben wir in dieser Dokumentation auf theoretische Details zu *GA* verzichtet, da hierzu genügend Literatur verfügbar ist. Dennoch muss auch hier kurz das Grundschema eines *GA* aufgezeigt werden, um anschließend direkt das Softwaredesign ableiten zu können.

	Population erzeugen Hier muss eine Grundpopulation erzeugt werden, mit der der *GA* arbeiten kann. Hierfür kann man verschiedene Ideen entwickeln, die wir als *Stategie* bezeichnen. Ein Beispiel für eine solche Strategie wäre: 1. doppelte Anzahl von Individuen mit zufälliger Chromosomenbelegung erzeugen 2. Berechnung der *Fitness* jedes dieser Individuen 3. Übernahme der "fittesten" Individuen in die

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.



Allgemeines Schema eines GA

Grundpopulation

Selektion

Bei der Selektion handelt es sich um ein Auswahlverfahren, welches die Individuen bestimmt, die in den sogenannten Heiratspool aufgenommen werden. Auch hier sind unterschiedliche Verfahren, also *Strategien* (z.B. paarweise zufällig auswählen) möglich.

Rekombination

In dieser Phase werden sogenannte *genetische Operatoren* auf die Phänotypen der Individuen im Heiratspool angewendet, um Kinder zu erzeugen, die im Laufe des Evolutionsprozesses immer bessere (Problemlösungs-) Eigenschaften entwickeln sollen. Hier sind wieder verschiedene Operatoren möglich, die auch als *Strategie* bezeichnet werden sollen.

Ersetzung

Da die Individuen im Allgemeinen nicht altern und sie somit bis zum Ende der Evolution in der Population vorhanden wären, muss man ein Verfahren wählen, welches die Population nach dem Erzeugen von Kindern wieder auf ein gewisses Maß reduziert. Der Einfachheit halber wird man wohl festlegen (wie wir dies auch getan haben), dass die Anzahl der Individuen einer Population konstant ist. Auch hier bestehen wieder mehrere Verfahren, dieses Ersetzungschema zu bewerkstelligen. Weiterhin wird nach diesem Schritt noch eine **Mutation** vorgenommen. Die Mutation ist ebenfalls ein *genetischer Operator*, welcher zufällig ausgewählte Gene manipuliert, um "neue Wege" in der Evolution einschlagen zu können. Die Mutation trägt also maßgeblich zur Entwicklung der Population über Generationen hinweg bei.

Ende?

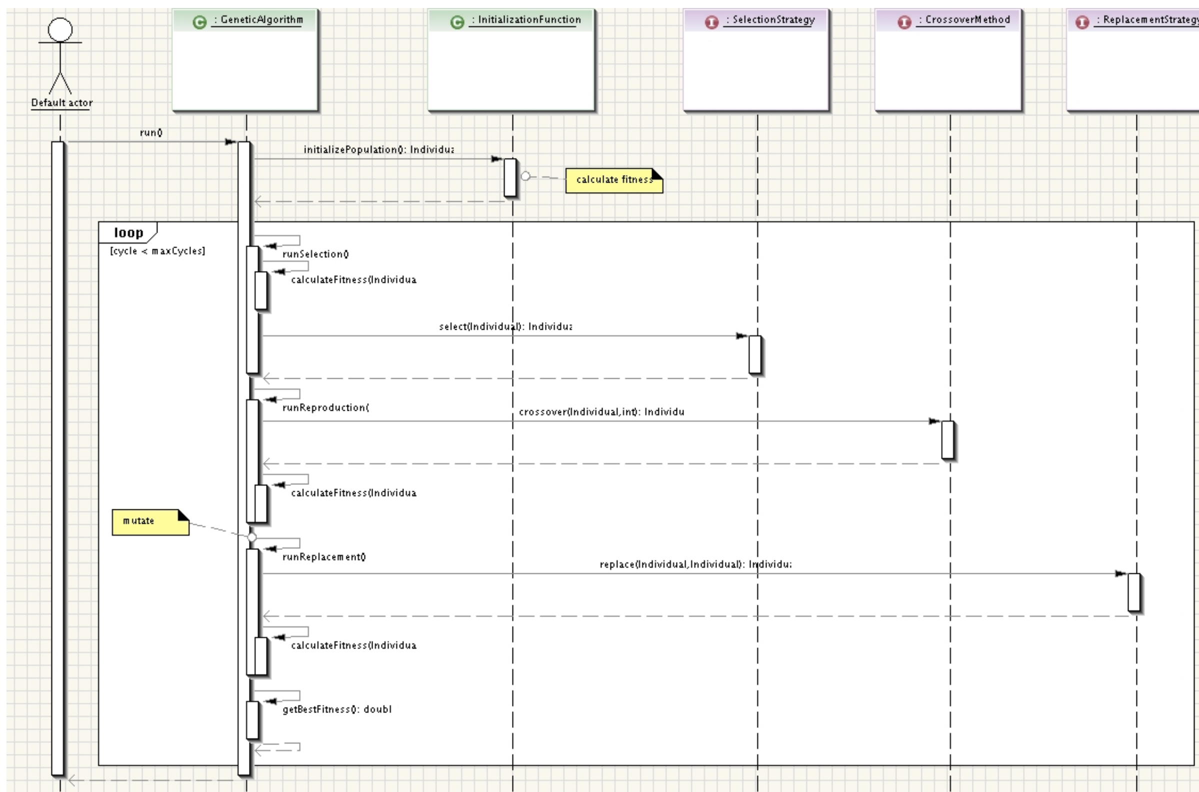
Es wird stets geprüft, ob die maximale Anzahl an Durchläufen erreicht ist oder die maximal benötigte *Fitness* erreicht wurde. In beiden Fällen wird der *GA* abgebrochen und das beste Individuum ausgegeben

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Wie man sich leicht vorstellen kann, benötigt man evtl. an einigen Stellen die *Fitness* der Individuen, je nachdem welche Verfahren/Strategien zur Anwendung kommen. Im Grundschema ist die Berechnung der *Fitness*, also das "Konvertieren" der Phänotypen in (hoffentlich) lebensfähige Individuen, an einer Stelle festgeschrieben. Da wir aber aus praktischen Gründen an verschiedenen Stellen diese Berechnung anstoßen, haben wir im Grundschema die *Fitnessberechnung* weggelassen (siehe auch [nächster Abschnitt]).

2.3.2. Modellierung des Ablaufes

Nachdem wir nun mit dem Ablaufschema eines *GA* vertraut sind, können wir diesen Prozess direkt als Sequenzdiagramm abbilden. Der hier aufgeführte Ablauf ist so tatsächlich in **GaFrame** implementiert.



Sequenzdiagramm der run-Methode

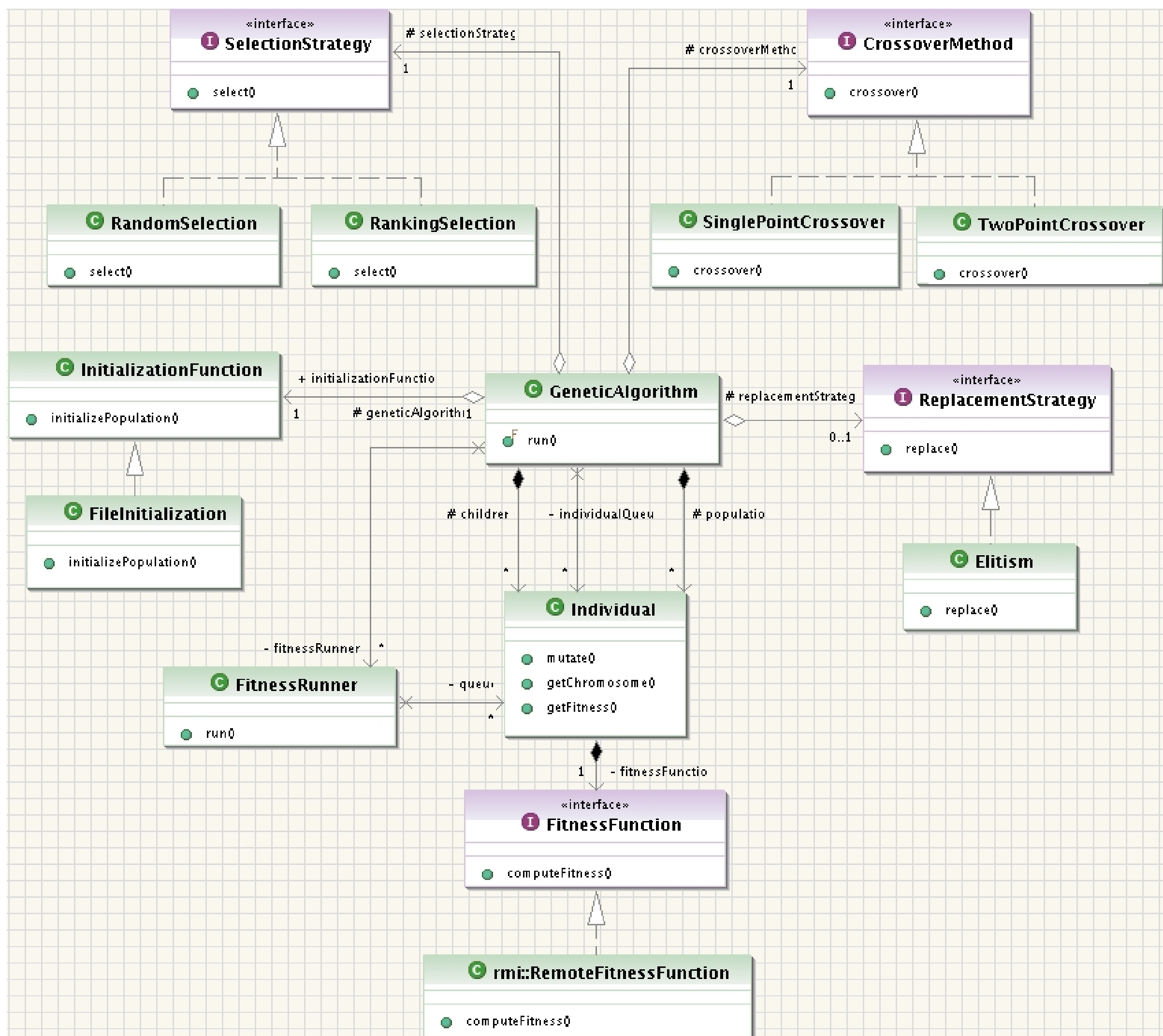
Wie man sieht, ist der komplette Ablauf des *GA* in einer Methode untergebracht, der `run()`-Methode, die von außen aufgerufen werden muss, um den *GA* in Gang zu setzen.

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Die einzelnen Teilschritte sind nochmals in eigene Methoden gekapselt, um Übersichtlichkeit und gute Lesbarkeit zu erreichen. In den jeweiligen Teilschritten werden schließlich Methoden von austauschbaren Strategien aufgerufen, die die entsprechenede Implementierung enthalten. Details hierfür finden sich im Abschnitt zur Softwarearchitektur.

2.3.3. Softwarearchitektur

2.3.3.1. Klassendiagramm



Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

GaFrame-Klassendiagramm

Das oben dargestellte Klassendiagramm enthält grundsätzlich alle Klassen, die das **GaFrame** beinhaltet. Lediglich einige Util-Klassen und der RMI-Server sind hier aus Platzgründen nicht mit aufgeführt. Wenn Sie sich für diese Util-Klassen interessieren, sollten Sie einen Blick in die javadoc-Dokumentation werfen und unter `de.htwdd.ga.util` nachschlagen. Auch das vollständige Klassendiagramm ist im Javadoc enthalten. In den folgenden Kapiteln werden einige Informationen zur verteilten Verarbeitung und RMI folgen, weswegen hier nur der Hinweis darauf erfolgt.

Wie man im Klassendiagramm sehr schön erkennen kann, ist **GaFrame** modular aufgebaut. Das bedeutet, dass es sich ohne Eingriffe in die bisherige Implementierung erweitern und anpassen lässt. Hiermit haben wir unseren definierten Anforderungen Rechnung getragen und konnten dies in parallel laufenden Belegen testen. Das Konzept hat sich in unseren Augen bewährt und lässt sich einfach handhaben.

2.3.3.2. Austauschbare Komponenten

Designziel war es, neben Effizienz des genetischen Prozesses, das System flexibel und allgemeingültig zu entwerfen. Um dies zu gewährleisten, haben wir eine einfache Form eines Extensionpoint-Mechanismus vorgesehen, der es gestattet, das System massiv zu beeinflussen, ohne nur eine Zeile Quellcode des eigentlichen **GaFrame**-Paketes ändern zu müssen.

Gelöst haben wir das Problem so, dass in den verschiedenen Phasen der `run()`-Methode verschiedene Aufgaben an verschiedene Objekte delegiert werden. Dieses Vorgehen ist im Sequenzdiagramm deutlich abzulesen.

Für den Austausch von Komponenten verfügt ein Objekt des Typs `GeneticAlgorithm` über Setter-Methoden um, die entsprechenden Objekte zu setzen. Wie aus dem Klassendiagramm ersichtlich, muss hierzu die gewünschte Schnittstelle genutzt und die Funktionalität ausprogrammiert werden. Im folgenden wird auf die einzelnen Bestandteile eingegangen.

Fitnessfunktion

Für jedes Problem muss eine individuelle Fitnessfunktion zur Verfügung gestellt werden. Hierzu haben wir das Interface `FitnessFunction` vorgesehen, welches die Methode `double computeFitness(BitSet chromosome)` vorschreibt. Hierbei verfügt jedes Individuum über eine eigene Fitnessfunktion, was nötig ist, um die parallele Nutzung zu gewährleisten.

Gesetzt wird die Fitnessfunktion über den Setter `public void setFitnessFunction(Class fitnessFunction)`. Da hier nur ein

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Class-Objekt übergeben wird, ist es uns möglich, mittels Reflection beliebig viele Instanzen der Fitnessfunktion zu erzeugen.

Initialisierungsfunktion und Mutation

Wir haben bereits zwei Implementierungen einer Initialisierungsfunktion dem Paket beigefügt. Nähere Informationen unter Verfahren.

Oft ist es nötig, für konkrete Anwendungsfälle spezielle Initialisierungsverfahren zu nutzen, um die Konvergenz des genetischen Prozesses zu beschleunigen oder sogar erst zu ermöglichen. Um eigene Implementierungen in **GaFrame** nutzen zu können, verfügt die Klasse `GeneticAlgorithm` über die Setter-Methode `void setInitializationFunction(InitializationFunction initializationFunction)`. Hiermit ist es möglich eine eigene, die Klasse `InitializationFunction` ableitende, Klasse zu nutzen.

Weiterhin sei erwähnt, dass es eigentlich nicht vorgesehen ist, die Klasse `Individual` zu verändern. Sollte dies unbedingt nötig sein, um zum Beispiel die **Mutation** grundsätzlich anders zu implementieren, dann können Sie in einer eigenen Initialisierungsfunktion Objekte Ihrer eigenen Idividuum-Klasse erzeugen. Bitte achten Sie hier allerdings darauf, dass das Individuum ein zentrales Element des Genetischen Algorithmus ist und Änderungen hier schnell dazu führen können, dass andere Verfahren nicht mehr ordungsgemäß auf den Individual-Objekten arbeiten können. **Erweitern Sie also bitte die Individual-Klasse nur, wenn Sie mit den Interna von GaFrame absolut vertraut sind!**

Selektionsverfahren

Die drei letzten Komponenten, die man austauschen kann, werden alle nach dem oben beschriebenen Schema eingehängt. Weiterhin ist hierfür jeweils ein Interface vorgesehen, um die Schnittstelle zu **GaFrame** zu definieren. Wir werden also die Beschreibung verkürzen und nur noch den Namen des Interfaces und der Setter-Methode nennen, um die entsprechende Strategieimplementierung in **GaFrame** einzuhängen.

Interface:

`SelectionStrategy`

Setter in `GeneticAlgorithm`:

`void setSelectionStrategy(SelectionStrategy selectionStrategy)`

Rekombinationsverfahren

Interface:

`CrossoverMethod`

Setter in `GeneticAlgorithm`:

`void setCrossoverMethod(CrossoverMethod crossoverMethod)`

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Ersetzungsstrategie

Interface:

`ReplacementStrategy`

Setter in `GeneticAlgorithm`:

`void setReplacementStrategy(ReplacementStrategy replacementStrategy)`

3. Implementierung

3.1. Implementierte Verfahren

Trotz Austauschbarkeit und Flexibilität ist es nicht sinnvol,l alle Standardverfahren, die im Gebiet der Genetischen Algorithmen bekannt und bewährt sind, für jeden Anwendungsfall neu zu schreiben. Aus diesem Grund haben wir einige dieser Verfahren direkt in **GaFrame** eingebaut, damit ein Nutzer möglichst schnell zu Ergebnissen gelangen kann.

Natürlich haben wir nicht alle uns bekannten Verfahren für die einzelnen Phasen implementiert, sondern nur diese, die uns als sinnvoll erschienen und die wir in den parallel laufenden Belegen nutzen wollten.

Im Folgenden wollen wir die gewählten Verfahren weiter erläutern.

3.1.1. Initialisierung

Zur Initialisierung des Evolutionsprozesses haben wir zwei Implementierungen beigefügt. Das Standardverfahren, welches verwendet wird, arbeitet nach dem Zufallsprinzip. Hierbei wird die Bit-Kombination des Chromosoms zufällig bestimmt und anschließend die Fitness für die Individuen berechnet. Weiterhin gestattet es eine Überinitialisierung, deren Grad mittels der Methode `void setInitializationCoeff(double initializationCoeff)` gesetzt werden kann. Als Standardwert ist hier 1.0 vorgegeben, was zu keiner Überinitialisierung führt. Dies hat den Grund, dass die Fitnessberechnung unter Umständen sehr lange dauern kann und ein zu optimistisch gewählter Wert das System extrem ausbremsen würde. Nach der Initialisierung werden dann die fittesten Individuen dem Evolutionsprozess zugeführt (entsprechend der Populationsgröße).

Das zweite Verfahren ist kein Initialisierungsverfahren im klassischen Sinne. Es stellt lediglich die Möglichkeit dar, eine als XML-Datei abgespeicherte Population zu laden, um einen vorangegangenen Evolutionsprozess weiter zu führen. Diese Klasse nutzt dabei die Funktionalität, welche durch eine Util-Klasse des **GaFrames** angeboten wird. Nähere Informationen finden Sie hierzu in der javadoc-Dokumentation für die Klasse `de.htwdd.ga.util.GaXmlUtil`.

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

3.1.2. Selektion

Auch für den Prozess der Selektion haben wir zwei Verfahren implementiert und dem **GaFrame** beigefügt. Die `RandomSelection` stellt eine einfache und performante Variante dar, die Paare von Heiratskandidaten zufällig bestimmt und auswählt. Dieses Verfahren sollte zum Experimentieren genutzt werden. Sind die Ergebnisse nicht befriedigend, sollten Sie die `RankingSelection` nutzen. Die rangbasierte Selektion hat eine höhere Laufzeitkomplexität, welche sich, je nach Problem, durch besserer Konvergenz relativieren kann.

3.1.3. Rekombination (Crossover)

Zum eigentlichen "verheiraten" der Individuen haben wir zwei einfache Crossover-Verfahren eingebaut. Standardmäßig wird `SinglePointCrossover` genutzt, es steht aber ebenfalls die Klasse `TwoPointCrossover` zur Verfügung. Eine Darstellung über die Wirkungsweise finden Sie unter "Optionale Konfigurationsmöglichkeiten" (Kreuzungsverfahren) im Abschnitt Nutzung.

3.1.4. Ersetzungsschema

Wir haben uns hier für den Elitismus entschieden und in der Klasse `Elitism` implementiert. Dieses Verfahren schien uns am sinnvollsten. Das Verfahren bildet eine Folgepopulation für die nächste Generation im Evolutionsprozess. Hierzu führt es die aktuelle Population (`population`) und die Kinder (`children`) zusammen. Natürlich unter Einhaltung der Populationsgröße, die den Einschränkungen entsprechend konstant sein soll.

3.1.5. Mutation

Die Mutation ist direkt in der Klasse `Individual` untergebracht und die entsprechende Methode des Individuums wird ausgeführt, nachdem das Ersetzungsschema die Folgepopulation erstellt hat. Nach dem Mutieren muss eventuell erneut geprüft werden, ob die Fitness der Individuen noch aktuell ist. Andernfalls muss sie neu berechnet werden.

Natürlich wird in **GaFrame** nicht jedes Individuum mutiert, sondern es wird mittels einer Mutationswahrscheinlichkeit angegeben, wie hoch die Wahrscheinlichkeit sein soll, dass ein Individuum mutiert werden soll. Wurde ein Individuum für die Mutation ausgewählt, wird zufällig ein Bit ermittelt und "umgekippt".

3.1.6. Graycodierung

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Es ist auch die Möglichkeit vorgesehen, die Chromosomen mittels einer Graycodierung zu verschlüsseln. Hierzu genügt es, der statischen Methode `void doGrayCoding(BitSet bitSet)` der Klasse `de.htwdd.ga.util.BitSetUtil` aufzurufen. Zum Entschlüsseln in der Fitnessfunktion können Sie das Chromosom mittels `void doInvGrayCoding(BitSet bitSet)` zurück transformieren.

3.2. Parallelisierung

Um einen Genetischen Algorithmus nutzbringend einsetzen zu können, ist es oft erforderlich, den Evolutionsprozess über einen relativ langen Zeitraum (eine große Anzahl von Zyklen) durchzuführen. Der Rechenaufwand zur Durchführung eines Genetischen Algorithmus kann sich dabei durch folgende Faktoren erhöhen:

- Große Populationen
- Große Chromosomen (eine hohe Chromosomenlänge)
- Langer Evolutionszeitraum (viele Zyklen)
- Komplexe, aufwändig zu berechnende Fitnessfunktionen

Da inzwischen viele PCs mit mehreren Prozessoren oder mit Mehrkernprozessoren ausgestattet sind, lag es nahe, für **GaFrame** direkt die Möglichkeit der Parallelverarbeitung vorzusehen.

3.2.1. Parallelisierung in GaFrame

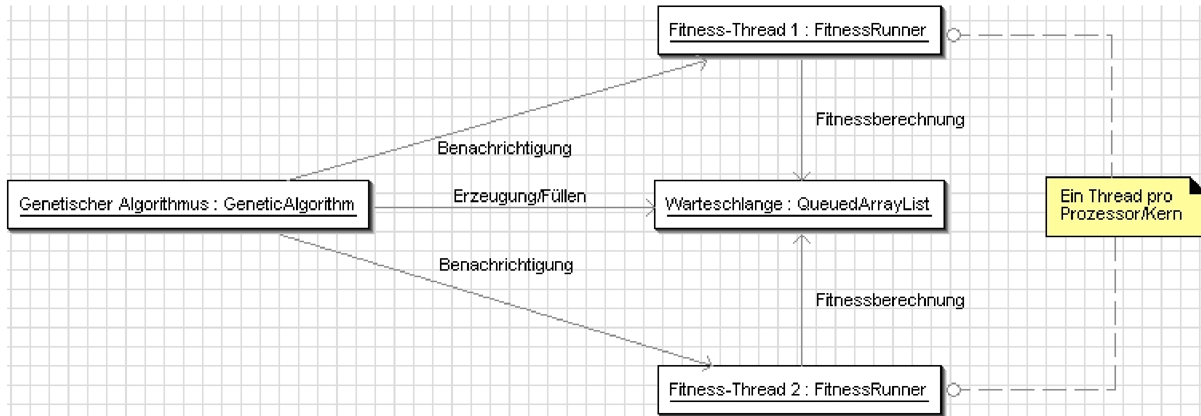
Um im Ablaufmodell von **GaFrame** parallele Verarbeitung effektiv nutzen zu können, musste zuerst ein geeigneter Erweiterungspunkt in der Architektur gefunden werden.

Die eigentlichen "genetischen Operationen" im Genetischen Algorithmus (Selektion, Rekombination, Mutation, Ersetzung) zeichnen sich durch konstanten Aufwand aus, der für die meisten mittels Genetischer Algorithmen zu lösenden Probleme etwa gleich sein sollte. Die Berechnung der Fitness eines Individuums ist aber bei jedem Problem individuell, hier können durchaus auch aufwändige Funktionen genutzt werden. Dies ist z.B. in den Belegen der Autoren der Fall. Aus diesem Grund entschieden wir uns, die Fitnessberechnung parallelisierbar zu gestalten. Sobald innerhalb des Genetischen Algorithums die Fitness einer Population zu berechnen ist (siehe Design), kann dies bei entsprechender Konfiguration auf mehrere Prozessoren bzw. Kerne verteilt werden. **GaFrame** nutzt hierbei pro Prozessor/Kern einen eigenständigen Thread, sodass die Kapazitäten der Maschine optimal ausgenutzt werden.

Das folgende Objektdiagramm zeigt nochmals den Zusammenhang: `GeneticAlgorithm` füllt, sobald eine Fitnessberechnung notwendig ist, die entsprechenden Individuen in eine Warteschlange. Die Threads, die die eigentliche Fitnessberechnung durchführen, werden

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

benachrichtigt, nehmen die Individuen aus der Warteschlange und berechnen ihre Fitness.



Objektdiagramm Multithreading

Das hier erläuterte Verfahren bildet auch die Basis für die im nächsten Abschnitt erläuterte verteilte Verarbeitung.

3.3. Verteilte Verarbeitung

In einigen vorstellbaren Szenarien kann auch die im vorherigen Abschnitt vorgestellte Möglichkeit der Parallelisierung die benötigte Rechenzeit nicht in "erträgliche" Dimensionen rücken. Wenn in einem solchen Fall dann (z.B. in einer Hochschulumgebung) außerdem noch eine große Anzahl an Rechnern zur Verfügung steht, kommt der Wunsch auf, den Genetischen Algorithmus nicht nur auf mehrere Prozessoren sondern gleich auf mehrere Rechner zu verteilen.

Auch diese Möglichkeit, die verteilte Verarbeitung (oder Clusterbetrieb), ist in **GaFrame** umgesetzt und kann direkt genutzt werden.

3.3.1. Verteilte Verarbeitung in GaFrame

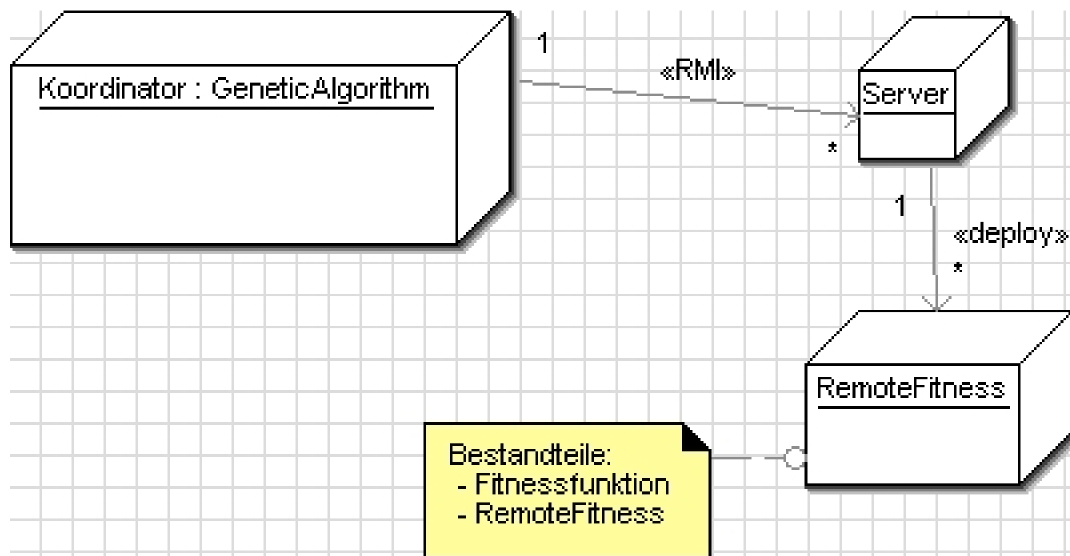
Aus den bereits erläuterten Gründen wird auch im Falle der verteilten Verarbeitung nur *die Berechnung der Fitnessfunktion* parallelisiert. Das angewendete Verfahren basiert auf der bereits vorgestellten Parallelisierung und erweitert diese, wobei die Möglichkeiten der *Remote Method Invocation* (RMI) genutzt werden.

Dabei tritt der Rechner, auf dem der Genetische Algorithmus gestartet wurde, als Koordinator auf und führt sämtliche Berechnungen bis auf Fitnessberechnungen selbst durch. Auf beliebig vielen anderen Rechnern laufen währenddessen RMI-Server, die bei Programmstart mit der zu nutzenden Fitnessfunktion initialisiert wurden.

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Als zusätzliches Feature sind auch die Serverprozesse weiter parallelisiert: Pro Server werden so viele Fitnessberechnungsthreads erzeugt, wie Prozessoren/Kerne vorhanden sind. Dadurch werden auch die entfernten Rechner optimal ausgelastet.

Im Verteilungsdiagramm wird die Situation deutlich: Eine Instanz von `GeneticAlgorithm` kann auf beliebig viele Server zugreifen, die wiederum mehrere Objekte vom Typ `RemoteFitness` zur Verfügung stellen.

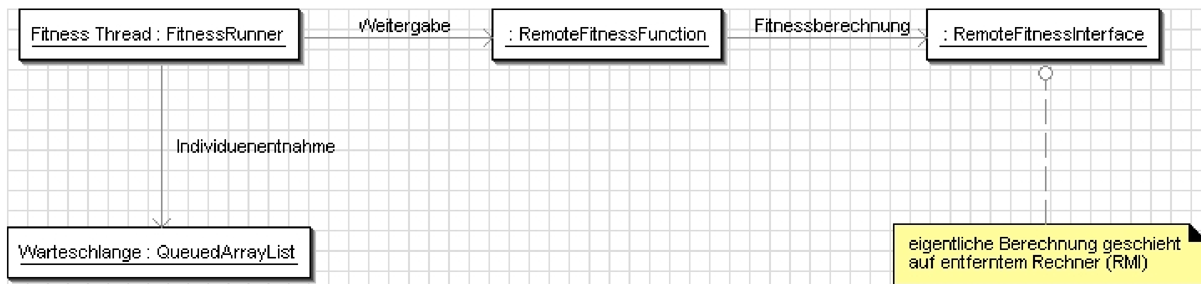


Verteilungsdiagramm für den Clusterbetrieb

Sobald Fitnesswerte zu berechnen sind, werden (wie bei der Parallelisierung) die entsprechenden Individuen in eine Warteschlange gefüllt. Die Fitness wird jetzt aber nicht lokal berechnet, sondern auf den entfernten Rechnern. Die Chromosomen der Individuen, deren Fitness zu berechnen ist, werden hierzu serialisiert und an die entsprechenden Server weitergegeben. Die Server führen dann die Berechnung durch und geben die Fitnesswerte an den Koordinator zurück.

Durch die Nutzung von RMI ist die Umsetzung der verteilten Verarbeitung transparent. Die entfernten Objekte (RemoteInterfaces) müssen lediglich beim Start des Genetischen Algorithmus initialisiert werden, im weiteren Programmverlauf ändert sich (gegenüber dem Betrieb in Parallelisierung) nichts. Das folgende Objektdiagram verdeutlicht dies nochmals:

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.



Objektdiagramm Clusterbetrieb

Mehraufwand entsteht allerdings dadurch, dass der Quellcode der Fitnesfunktion sowie die **GaFrame**-Bibliothek auf die Server verteilt und die Serverprozesse dort gestartet werden müssen. Hinweise hierfür finden sich unter Verteilte Verarbeitung.

Sollte im Verlauf des Genetischen Algorithmus einer der entfernten Server ausfallen, läuft der Genetische Algorithmus trotzdem weiter (solange noch mindestens 1 Server zur Verfügung steht), der ausgefallene Server wird einfach ignoriert.

4. Nutzung

4.1. Nutzung von GaFrame

Note:
Die zur Nutzung von **GaFrame** notwendige Bibliothek wird im Downloadbereich zur Verfügung gestellt.

Im folgenden Abschnitt wird ein grundlegendes "Kochrezept" zur Erstellung eines eigenen Genetischen Algorithmus vorgestellt.

Dabei geht es zuerst nur um die lokale Nutzung, die Möglichkeiten der parallelen Nutzung bzw. der Nutzung in einem Cluster werden auf eigenen Seiten erläutert.

Um die Bibliothek in einer eigenen Anwendung nutzen zu können, muss `ga-frame.jar` im Classpath enthalten sein. Weitere Konfigurationen sind nicht nötig, werden die Features zum Speichern/Laden von Populationen genutzt, muss auch die Bibliothek JDOM im Classpath liegen.

`ga-frame.jar` enthält neben den class-Dateien auch die Quelldateien des Projekts, so dass der Quellcode (und damit auch javadoc) zugänglich sind. Details zum Design des Frameworks sind auch aus Design und Javadoc zu entnehmen.

4.1.1. Das Rucksackproblem

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Ein Beispiel für einen einfachen Genetischen Algorithmus ist die Klasse `de.htwdd.ga.examples.KnapSack`, diese ist direkt ausführbar und modelliert ein klassisches Beispiel Genetischer Algorithmen, das Rucksackproblem.
Das Problem besteht darin, aus einer Menge verschiedener Objekte (z.B. Zubehör für eine Bergwanderung) diejenigen auszuwählen, die in einen Rucksack gepackt werden sollen. Jedem Gegenstand ist dabei ein Nutzwert und ein Gewicht zugeordnet, der Rucksack hat ein maximal zulässiges Gesamtgewicht. Die optimale Kombination ist zu ermitteln.

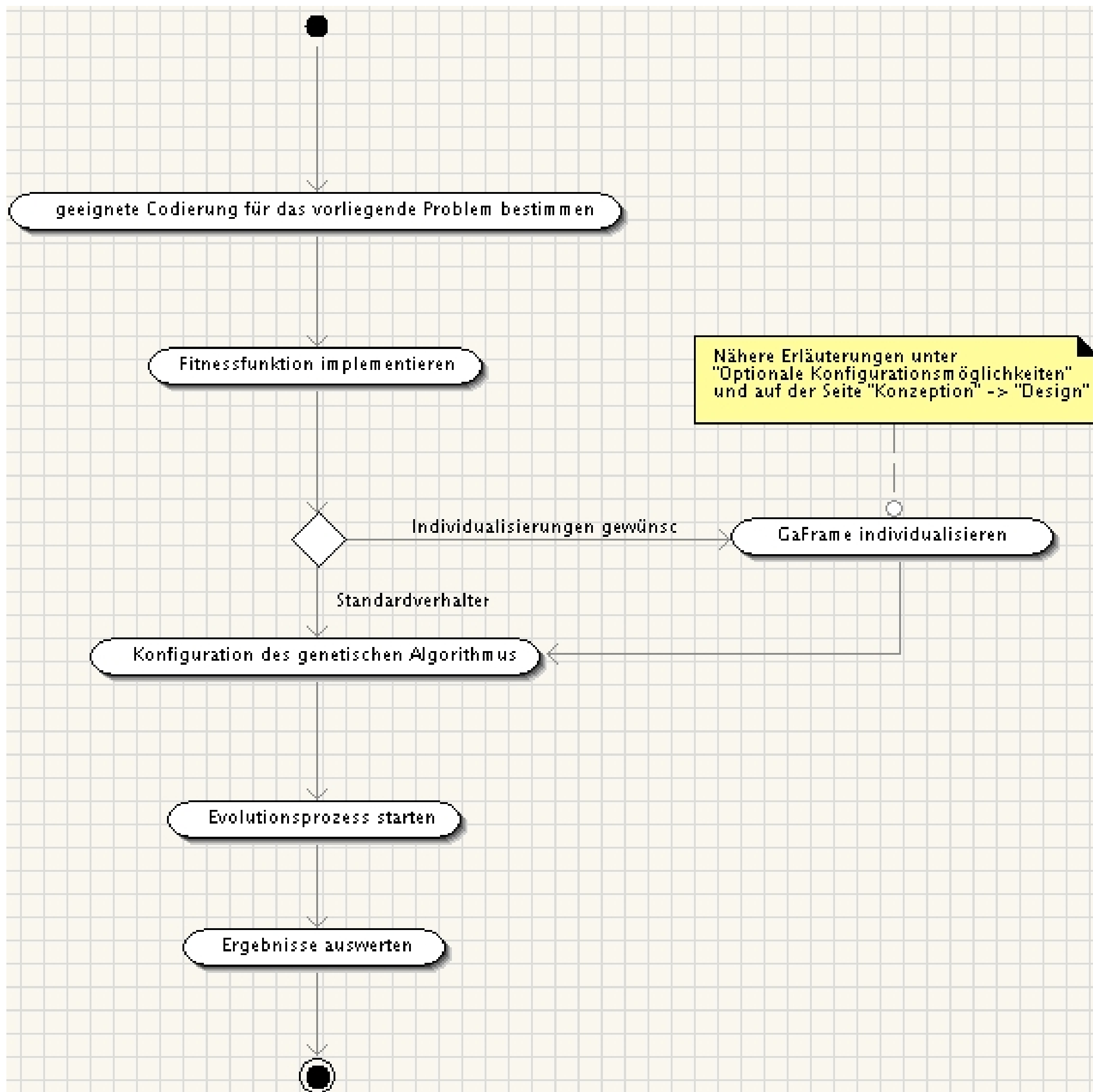
Anhand dieses Beispiels wird die Nutzung von **GaFrame** deutlich, die wichtigsten Elemente des Frameworks werden hierzu im nächsten Abschnitt erläutert.

Note:
Eine einfache Art und Weise, einen eigenen Genetischen Algorithmus zu erstellen, ist also, die genannte Klasse zu kopieren und sich durch Modifikationen mit dem Framework vertraut zu machen.

4.1.2. Kochrezept für einen Genetischen Algorithmus

Das folgende Diagramm zeigt das grundsätzliche Vorgehen, um mit **GaFrame** einen eigenen Genetischen Algorithmus zu implementieren. Die einzelnen Punkte werden in den folgenden Unterabschnitten weiter erläutert.

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.



Kochrezept

4.1.2.1. Grundlegende Elemente von GaFrame

Chromosom

Ein Chromosom besteht aus einer Folge von Bits, in **GaFrame** wird es durch ein

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

`java.util.BitSet` repräsentiert.

Vor der Nutzung eines Genetischen Algorithmus muss also festgelegt werden, wie sich das vorliegende Problem am sinnvollsten codieren lässt. Eine sinnvolle Codierung des Rucksackproblems ist, für jeden Gegenstand ein Bit zu reservieren, das festlegt, ob der Gegenstand im Rucksack enthalten ist oder nicht.

Der folgende Code gibt eine "Rucksackpopulation" auf der Konsole aus. Jeder im Rucksack enthaltene Gegenstand wird mit einem `x` markiert.

```
private static void displayPopulation(Individual[] population)
{
        StringBuffer out = new StringBuffer();
        for (Individual individual : population)
        {
                BitSet b = individual.getChromosome();

                out.append("individual ");
                for (int i = 0; i < KnapsackFitness.NUM_ITEMS; i++)
                {
                        if (b.get(i))
                                out.append("x ");
                        else
                                out.append("- ");
                }
                out.append("\n");
        }
        System.out.println(out.toString());
}
```

Bei der Arbeit mit einem BitSet kann entweder direkt auf die einzelnen Bits zugegriffen werden (wie im Beispiel), oder es kann die Klasse `de.htwdd.ga.util.BitSetUtil` genutzt werden, um Ganz- und Gleitkommazahlen in bestimmte Teile des BitSets zu schreiben bzw. wieder auszulesen.

Der folgende Code extrahiert eine Ganzzahl aus den Bits 3-12 des BitSets *chromosome*:

```
BitSetUtil.bitSetToInt(chromosome, 3, 10);
```

An dieser Stelle wollen wir auch darauf hinweisen, dass standardmäßig keine Gray-Codierung verwendet wird. Diese kann allerdings leicht durch die eben schon genutzte Klasse `de.htwdd.ga.util.BitSetUtil` nachgereicht werden. Hierzu muss man nur nach dem Setzen der Bitkombination `BitSetUtil.doGrayCoding(bitSet)` aufrufen. Diese statische Methode transformiert das übergebene `BitSet` (in binärer Darstellung) in einen Gray-Code.

Die Rücktransformation kann auf gleiche Weise mittels `BitSetUtil.doInvGrayCoding(BitSet bitSet)` erfolgen.

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Individuen/Populationen

Ein Individuum (`de.htwdd.ga.Individual`) wird duch sein Chromosom repräsentiert, eine Population besteht aus einem Array von Individuen.

Die Klasse `de.htwdd.ga.util.GaXMLUtil` bietet Methoden, um eine Population in einer XML-Datei zu persistieren oder sie wieder zu extrahieren.

Fitnessfunktion

Um die Fitness eines Individuums zu bestimmen, benötigt der Genetische Algorithmus eine auf das Problem zugeschnittene Fitnessfunktion. Diese muss das Interface `de.htwdd.ga.FitnessFunction` implementieren.

Die Fitness eines Rucksackindividuums wird duch die aufsummierten Nutzwerte der enthaltenen Gegenstände bestimmt, die Nutzwerte werden aus einer Tabelle ausgelesen. Ein zu schwerer Rucksack hat die Fitness 0. Das Array `knapSackArray` enthält Nutzwert und Gewicht der einzelnen Gegenstände, der zweite Gegenstand hat z.B. das Gewicht 7 und den Nutzwert 5.

```
public class KnapsackFitness implements FitnessFunction
{
    public static final int NUM_ITEMS   = 9;
    private static final int MAX_WEIGHT = 58;
    public static long delay            = 0;

        private int knapSackArray[][] = {
        { 3, 3 }, { 7, 5 }, { 4, 2 }, { 12, 11 },
        { 8, 4 }, { 10, 6 }, { 9, 2 }, { 14, 15 },
        { 10, 12 }, { 12, 9 } };



        public double computeFitness(BitSet chromosome)
        {
                int weightSum = 0, valueSum = 0;
                for (int iGene = 0; iGene < NUM_ITEMS; ++iGene)
                {
                        if (chromosome.get(iGene))
                        {
                                weightSum += knapSackArray[iGene][0];
                                valueSum += knapSackArray[iGene][1];
                        }
                }

                if (weightSum <= MAX_WEIGHT)
                        return valueSum;
                else
                        return 0;
        }
}
```


Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Genetischer Algorithmus

Die Klasse `de.htwdd.ga.GeneticAlgorithm` ist die zentrale Klasse von **GaFrame**, sie steuert den Ablauf des Genetischen Algorithmus. Um einen Genetischen Algorithmus zu nutzen, muss diese Klasse instanziiert werden. Dabei ist anzugeben:

- `chromosomeLength` - die Länge des Chromosoms aller Individuen
- `maxCycles` - die maximale Anzahl von Generationen, die erzeugt werden sollen
- `terminationFitness` - sobald mindestens ein Individuum der Population diese Fitness erreicht, wird der Algorithmus beendet.
- `populationSize` - die Anzahl von Individuen pro Population

Im Weiteren muss dem Genetischen Algorithmus eine Fitnessfunktion (siehe unten) angegeben werden. Optional können außerdem die einzelnen Bestandteile (Initialisierungsfunktion, Ersetzungsschema, ...) definiert werden, falls von den Standardeinstellungen abweichende Verfahren verwendet werden sollen.

`run()` startet den Genetischen Algorithmus.

Der folgende Code zeigt die entsprechende Lösung im Rucksackproblem:

```
GeneticAlgorithm algorithm = new
GeneticAlgorithm(KnapsackFitness.NUM_ITEMS, 100, 66, 20);
algorithm.setFitnessFunction(KnapsackFitness.class);
algorithm.setCrossoverMethod(new TwoPointCrossover());
algorithm.setMutationRate(0.1);
algorithm.setSelectionStrategy(new RankingSelection(.3, .4));

algorithm.run();
```

4.1.2.2. Optionale Konfigurationsmöglichkeiten

GaFrame individualisieren

GaFrame kann leicht individualisiert werden, um z.B. Log-Ausgaben oder das automatische Speichern der Populationen nach jedem Durchlauf zu realisieren. Zu diesem Zweck muss lediglich die Klasse `de.htwdd.ga.GeneticAlgorithm` abgeleitet werden. Alle nicht als final deklarierten Methoden lassen sich leicht überschreiben, um individuelle Erweiterungen zu realisieren. Dabei ist darauf zu achten, dass im Verlauf der Methode `super` aufgerufen wird.

Im folgenden Beispiel wird die Methode `runReplacement()` überschrieben, um eine Logausgabe zu realisieren und die Population zu speichern.

```
public class MyGA extends GeneticAlgorithm
{
```


Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

```
        public MyGA(int chromosomeLength, int numCycles, int
populationSize)
        {
                super(chromosomeLength, numCycles, 0.98,
populationSize);
        }

        @Override
        protected void runReplacement()
        {
                super.runReplacement();

                // Fitnesswerte aller Individuen ausgeben
                StringBuffer buffer = new StringBuffer();
                buffer.append("##population fitness after cycle ");
                buffer.append(cycle + 1);
                IndividualSorter.sortIndividuals(population);
                for (Individual individual : population)
                {
                        buffer.append(String.format(" %1.4f",
individual.getFitness()));
                }
                System.out.println(buffer.toString());

                //Population speichern
                GaXmlUtil.writeToFile(population, "data/" + cycle +
".xml");
        }
}
```

Neben der Möglichkeit, `GeneticAlgorithm` zu überschreiben, können auch alle weiteren austauschbaren Komponenten durch individuelle Implementierungen (z.B. eine eigene Ersetzungsstrategie) ersetzt werden.

Mutationsrate

Die Mutationsrate bestimmt, mit welcher Wahrscheinlichkeit ein Individuum mutiert, also sein Erbmaterial zufällig verändert. Mutationen werden nach der Ersetzung durchgeführt (siehe Ablaufmodellierung).

Beispiel:

```
GeneticAlgorithm algorithm;
...
algorithm.setMutationRate(0.2);
```

Initialisierungsfunktion

Die Initialisierungsfunktion erzeugt eine Startpopulation, mit der der Genetische Algorithmus im Folgenden arbeiten wird (`de.htwdd.ga.InitializationFunction`). Standardmäßig werden die hier

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

erzeugten Individuen mit einem zufällig erzeugten Chromosom ausgestattet. Weiterhin ist es möglich, eine Überpopulation erzeugen zu lassen, deren "fitteste" Individuen dann die initiale Population bilden. Standardmäßig wird keine Überpopulation erzeugt.

Das folgende Beispiel erzeugt eine Überpopulation von 50%:

```
GeneticAlgorithm algorithm;
...
algorithm.getInitializationFunction().setInitializationCoeff(1.5);
```

Die Klasse `de.htwdd.ga.FileInitialization` erlaubt es, eine aus einer XML-Datei extrahierte Population als Startpopulation zu nutzen, weitere eigene Implementierungen sind denkbar.

Beispiel:

```
GeneticAlgorithm algorithm;
...
FileInitialization init = new FileInitialization("alte_Population.xml");
algorithm.setInitializationFunction(init);
```

Selektionsverfahren

Das Selektionsverfahren bestimmt, welche Individuen einer aktuellen Population sich vermehren dürfen, Basisklasse für Selektionsverfahren ist `de.htwdd.ga.SelectionStrategy`.

Vorhandene Selektionsverfahren:

- zufällige Auswahl von Heiratskandidaten: `de.htwdd.ga.RandomSelection`
- rangbasierte Auswahl: `de.htwdd.ga.RankingSelektion`

Beispiel:

```
GeneticAlgorithm algorithm;
...
algorithm.setSelectionStrategy(new RankingSelection(.3, .4));
```

Kreuzungsverfahren

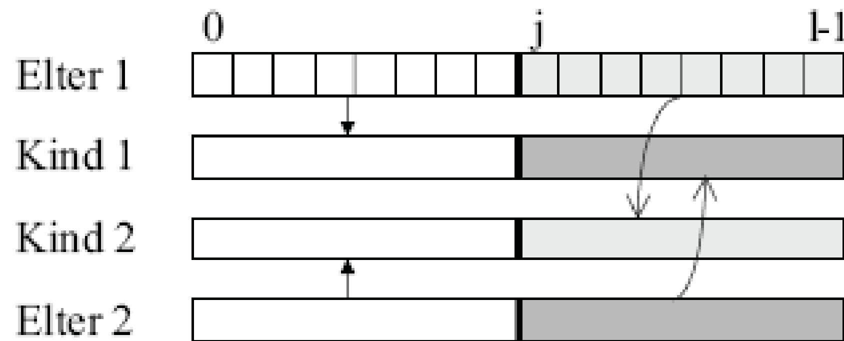
Um aus einer vorhandenen Population von Individuen Nachkommen zu gewinnen, werden jeweils zwei Individuen aus dem Heiratspool mittels eines *Kreuzungsverfahrens* rekombiniert.

Grundsätzlich bedeutet dies, dass das "Erbgut" (also das Chromosom) der Eltern in geeigneter Weise gespalten wird, wodurch man (üblicherweise) zwei neue Individuen, die Kinder, erhält.

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

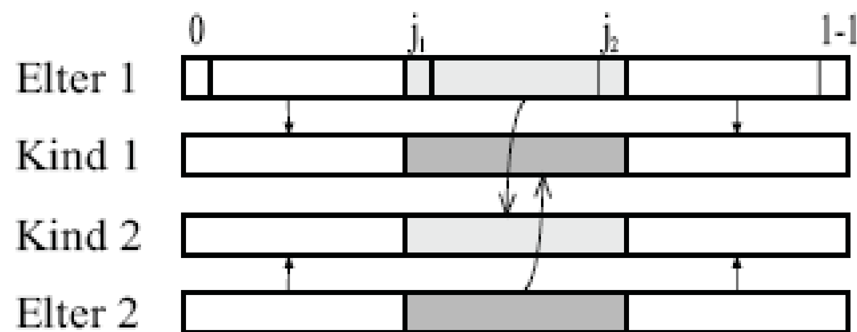
GaFrame enthält zwei grundlegenden Kreuzungsverfahren:

- `de.htwdd.ga.SinglePointCrossover`: Bei diesem Verfahren wird zufällig eine Stelle des Chromosoms ausgewählt, die Chromosomen der beiden Eltern werden an dieser Stelle aufgeteilt. Durch Vertauschen der jeweils zweiten Hälften der Chromsomen entstehen zwei neue Individuen.



Ein-Punkt-Kreuzung

- `de.htwdd.ga.TwoPointCrossover`: Aus dem Namen wird bereits klar, dass bei diesem Verfahren zwei Stellen des Chromosoms ausgewählt werden. So wird aus der Mitte der Chromosomen der Eltern ein Teil herausgetrennt, durch Kreuzung erhält man auch hier zwei neue Individuen.



Zwei-Punkt-Kreuzung

Beispiel:

```
GeneticAlgorithm algorithm;
...
algorithm.setCrossoverMethod(new TwoPointCrossover());
```


Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

4.2. Unterstützung für Multithreading

Oft ist es sinnvoll, **GaFrame** im MultiThreaded-Modus zu betreiben. Weitere Erläuterungen dazu finden sich im Abschnitt Implementierung.

4.2.1. Nutzung von Multithreading

> **Warning:**
> Multithreading ist standardmäßig nicht aktiviert und sollte nur genutzt werden, wenn die Fitnessfunktion des vorliegenden Problems aufwändig zu berechnen ist im Verhältnis zum rechnerischen Mehraufwand, der durch die Threadsynchronisation entsteht. Außerdem ist Multithreading nur bei Mehrprozessor- oder Mehrkernmaschinen sinnvoll.
> Sind diese Bedingungen erfüllt, bringt der Einsatz von Multithreading allerdings einen enormen Zeitgewinn.

Multithreading wird einfach durch das Setzen der Property `multiThreaded` der Klasse `de.htwdd.ga.GeneticAlgorithm` aktiviert:

```
GeneticAlgorithm algorithm = new
GeneticAlgorithm(KnapsackFitness.NUM_ITEMS, 100, 66, 20);
...
algorithm.setMultiThreaded(true);
algorithm.run();
```

Das Beispiel stammt aus dem schon im vorherigen Abschnitt erläuterten Beispielprogramm zum Rucksackproblem.

Zu beachten ist, dass pro Thread nur eine Instanz der Fitnessfunktion erzeugt wird, die dann im weiteren Verlauf wiederverwendet wird.

4.3. Verteilte Verarbeitung (Clusterbetrieb)

Sind aufwändige Fitnessfunktionen zu berechnen und steht ein Rechnerpool zur Verfügung, kann **GaFrame** mit relativ geringem Aufwand für den Clusterbetrieb eingerichtet werden. Diese Seite erläutert die Vorgehensweise an einem Beispiel.

4.3.1. Konfiguration für Clusterbetrieb

Die eigentliche Konfiguration des Genetischen Algorithmus erfordert keinen großen Aufwand. Es muss lediglich ein (String-) Array mit Servernamen übergeben werden. `GeneticAlgorithm` versucht dann, zu jedem der angegebenen Server eine RMI-Verbindung aufzubauen und erzeugt so viele entfernte Prozesse, wie auf dem jeweiligen Hostsystem Prozessoren/Kerne vorhanden sind. Diese Prozesse werden dann mit je einer Instanz der zu nutzenden Fitnessfunktion initialisiert.

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Beispiel:

```
GeneticAlgorithm algorithm;
...
String servers[] = {"isys23", "isys24", "isys25"};
algorithm.activateRMI(servers);
algorithm.run;
```

Sinnvollerweise würde man die Servernamen im realen Fall aus einer Datei auslesen.

4.3.2. Koordination der RMI-Server

Das eigentliche Problem des Clusterbetriebs besteht in der Koordination der Server. Am einfachsten lassen sich die auftretenden Probleme lösen, wenn man per SSH und PublicKey-Authentifizierung (möglichst ohne Passwort) auf die verschiedenen Server zugreifen kann. Die Codeverteilung kann einfach mittels SCP erfolgen, das Starten und Beenden der Serverprozesse lässt sich z.B. mit einfachen Bashskripten realisieren.

Die folgenden Beispiele gehen davon aus, dass sich der Quellcode, die **GaFrame**-Bibliothek und die genannten Skripte auf den jeweiligen Zielrechnern im "gleichen" Verzeichnis (also z.B. `~/ga`) befinden.

Die Servernamen werden als Liste in der Datei `serverList` hinterlegt:

```
isys23 isys24 isys25
```

Das Skript `startServers.sh` wird z.B. mit der Zeile ``startServers.sh `cat serverList` `` aufgerufen und startet dann auf jedem der Server ein weiteres Skript, dem es das aktuelle Verzeichnis als Parameter übergibt:

```
dir=`pwd`

for server in `cat serverList`
do
        echo $server
        ssh $server $dir/rmiStart.sh $dir &
done
```

Das Skript `rmiStart.sh` wird auf jedem entfernten Rechner ausgeführt, startet die RMI-Registry und registriert die Klasse `RemoteFitness` für den RMI-Zugriff. Damit ist der "Cluster" betriebsbereit.

```
cd $1
cd bin
pwd
```


Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

```
rmiregistry &
sleep 1
java -cp . de.htwdd.ga.rmi.RemoteFitness
```

4.3.3. Beenden der RMI-Server

Nach dem Durchlauf des Genetischen Algorithmus lassen sich die Server z.B. mit folgendem Skript (`stopServers.sh`), das mit `` stopServers.sh `cat serverList` `` aufgerufen wird, beenden:

```
for server in `cat serverList`
do
        echo $server
        ssh $server killall java
        ssh $server killall rmiregistry
done
```

Die Verwendung von `killall` ist zugegebenermaßen sehr radikal, aber das Beispiel kann (und sollte) individuell angepasst werden.

5. Downloads

Note:
Um **GaFrame** nutzen zu können, wird eine Java Virtual Machine 1.5.x (oder höher) benötigt. Zum Übersetzen der Quellen sind ein entsprechendes JDK sowie das Buildwerkzeug Ant notwendig.
JavaSE ist z.B. unter java.sun.com erhältlich.
Ant kann unter ant.apache.org heruntergeladen werden.
Um die Möglichkeit der Persistierung in XML-Dateien nutzen zu können, wird die Bibliothek JDOM benötigt.

5.1. Binärdistribution

Die Binärdistribution von **GaFrame** steht als JAR-File zum Download bereit. Weitere Informationen zur Nutzung können der Entwicklerdokumentation und dem Abschnitt Nutzung dieser Website entnommen werden.

- ga-frame.jar

5.2. Quelldistribution

Die Quelldistribution enthält neben den Java-Quellen und benötigten Bibliotheken die Entwicklerdokumentation (Javadoc), die auch UML-Klassendiagramme enthält.

Um **GaFrame** zu kompilieren, muss nach dem Entpacken der ZIP-Datei im Verzeichnis *ga-frame* der Befehl `ant compile` ausgeführt werden, `ant run` startet eine

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.

Beispielapplikation.

Die ZIP-Datei enthält außerdem eine Eclipse-Projektdatei, sodass das Projekt auch direkt mit Eclipse bearbeitet werden kann.

- ga-frame_src.zip

5.3. Subversion-Repository

Das Projekt wird mittels des Versionsmanagementsystems Subversion verwaltet.

Das entsprechende Repository ist unter http://www.hoelzware.de/svn/gaFrame/ erreichbar, der Zugriff kann sowohl per Browser als auch über einen SVN-Client (anonymer Zugriff) erfolgen.

6. Entwicklerdokumentation

Javadoc wird durch ant integriert und ist nicht in der PDF-Version enthalten

7. Export

Copyright © 2007 Dominik Hölz, Wolfram Wagner All rights reserved.