1 Implementierung des Frameworks

Implementieren Sie im Paket *ga.framework* eine Klasse *GeneticAlgorithm*, die den oben beschriebenen Ablauf eines genetischen Algorithmus mithilfe der gegebenen Interfaces und Klassen umsetzt.

1.1 Grundimplementierung (Spezifikation) [4 Punkte]

Um ein Optimierungsproblem mithilfe der Klasse *GeneticAlgorithm* lösen zu lassen, müssen von einem Nutzer folgende Parameter angegeben werden:

- das Problem, welches gelöst werden soll (Klasse *Problem*),
- die Größe der zu verwendenden Population,
- eine Liste evolutionärer Operatoren, die zur Suche neuer Lösungen verwendet werden sollen (Interface *EvolutionaryOperator*),
- die Fitnessfunktion, die zur Bestimmung der Qualität einer Lösung herangezogen werden soll (Interface *FitnessEvaluator*),
- der Überlebensoperator (Interface *SurvivalOperator*) und
- als Terminierungskriterium die Anzahl der durchzuführenden Iterationen/Evolutionsschritte.

Über eine Methode *runOptimization()* soll der Nutzer die Optimierung durchführen können. Der Ablauf soll dabei wie folgt aussehen:

- Entsprechend der festgelegten Populationsgröße wird zunächst eine Startpopulation erstellt. Die Erstellung einzelner Lösungen wird dabei durch die Methode *createNewSolution()* der konkreten *Problem*-Implementierung übernommen. Über die Methode *evaluate()* des vom Nutzer festgelegten *FitnessEvaluators* wird für alle erstellten Lösungen die Fitness bestimmt.
- Aus der vom Nutzer angegebenen Liste evolutionärer Operatoren wird in jeder Iteration zufällig *einer* gewählt. Dieser wird, jeweils durch Aufruf der Methode *evolve()*, auf jedes Element der aktuellen Population einmal angewandt und erzeugt dabei einen Nachkommen.

- Die Fitness der Nachkommen wird wieder über die Methode *evaluate()* des *FitnessEvaluators* bestimmt und die Nachkommen werden der aktuellen Population hinzugefügt.
- Der vom Nutzer angegebene *SurvivalOperator* wählt aus dieser erweiterten Population abschließend mit der Methode *selectPopulation()* die Population für die nächste Iteration aus.
- Es werden solange neue Iterationen durchgeführt, bis das vom Nutzer angegebene Limit erreicht ist.
- Tritt ein Fehler auf, wird die Optimierung abgebrochen und eine passende Fehlermeldung ausgegeben.
- Konnte die Optimierung ohne Auftreten von Fehlern durchgeführt werden, soll von der Methode *runOptimization()* die in der letzten Iteration entstandene Population zurückgegeben werden.

1.2 Überlebensoperator [2 Punkte]

Implementieren Sie einen einfachen Überlebensoperator *TopKSurvival*. Dieser soll, aus der übergebenen Liste von Lösungen, eine Population mit der vom Nutzer festgelegten Populationsgröße auswählen. Dabei sollen auf jeden Fall die besten k Lösungen (mit der höchsten Fitness) übernommen werden. Ist die verwendete Populationsgröße größer als k, werden zufällig weitere Lösungen aus der übergebenen Liste ausgewählt, bis eine Population mit der entsprechenden Populationsgröße erreicht ist. Lösungen dürfen dabei auch mehrmals in der vom Selektionsoperator erstellten Population vorkommen. Ist k größer als die verwendete Populationsgröße, soll eine *SurvivalException* geworfen werden.

1.3 Selektionsoperator [3 Punkte]

Üblicherweise wird in einer Iteration nicht jedes Element der Population weiterentwickelt. Stattdessen sollten qualitativ gute Lösungen durch einen sogenannten Selektionsoperator häufiger die Möglichkeit erhalten, Nachkommen zu produzieren. In unserem Fall soll ein Selektionsoperator genau ein Elternelement aus der aktuellen Population wählen, auf dem dann ein zufällig gewählter evolutionärern Operator angewandt wird.

Dies wird solange wiederholt, bis die Anzahl der Nachkommen der Populationsgröße entspricht.

Überlegen Sie sich ein sinnvolles Interface für Selektionsoperatoren, setzen Sie dieses in einer Interface-Klasse *SelectionOperator* um und binden Sie diese geeignet in den Ablauf des genetischen Algorithmus aus Aufgabe 1.1 ein. Implementieren Sie dann auf Basis des Interfaces *SelectionOperator* in einer Klasse *TournamentSelection* eine sogenannte Tournament-Selection. Diese soll aus der Population zunächst zwei zufällige Lösungen auswählen. Die Auswahl erfolgt *mit Zurücklegen*, d.h., die gleiche Lösung kann auch mehrmals gewählt werden. Zurückgegeben wird die Lösung mit der höheren Fitness bzw. die zuerst gewählte Lösung, falls beide Lösungen die gleiche Fitness besitzen.

1.4 Erweiterung um Fluent-API [4 Punkte]

Verändern Sie Ihre Implementierung der Klasse *GeneticAlgorithm* so, dass diese ein Fluent-Interface[5] zur Spezifikation der in Aufgabe 1.1 genannten Parameter anbietet. Im Folgenden ist beispielhaft ein entsprechender Aufruf einer Optimierung dargestellt.

```
1 GeneticAlgorithm ga = new GeneticAlgorithm();
2 List<Solution> result = ga.solve(yourProblem)
3         .withPopulationOfSize(10)
4         .evolvingSolutionsWith(yourEvolutionaryOperator)
5         .evolvingSolutionsWith(yourEvolutionaryOperator)
6         .evaluatingSolutionsBy(yourFitnessEvaluator)
7         .performingSelectionWith(yourSelectionOperator)
8         .stoppingAtEvolution(100)
9         .runOptimization();
```

Beachten Sie, dass Ihr *Fluent-Interface* die im Beispiel dargestellte Reihenfolge der Parameter erzwingen sollte. Zudem soll eine Optimierung erst gestartet werden können, wenn alle nötigen Parameter festgelegt wurden. Bedenken Sie auch, dass die Angabe mehrerer evolutionärer Operatoren möglich sein muss.

Tipp: Die Umsetzung des Fluent-Interfaces lässt sich durch mehrere innere Klassen realisieren, von denen jede jeweils für einen Parametertyp verantwortlich ist.

[5] https://de.wikipedia.org/wiki/Fluent_Interface

2 Das Rucksack-Problem

Ein beliebtes Beispiel für Optimierungsprobleme ist das sogenannte Rucksack-Problem[6]. Für dieses sollen Sie nun eine zum Framework aus Aufgabe 1 passende Implementierung schreiben, die es ermöglicht, konkrete Instanzen des Rucksackproblems mithilfe des Frameworks zu lösen.

Die Implementierung von Optimierungsproblemen soll dabei in einem separaten IntelliJ-Modul erfolgen. Zudem soll dieses, genau wie das Framework, als Modul im Sinne von JPMS entwickelt werden. Es ist dabei wichtig zu beachten, dass JPMS und das IntelliJ-Modulsystem zwei unabhängige Systeme zur Verwaltung von Modulen darstellen. Um ein korrektes Zusammenspiel zwischen Framework- und Problemimplementierung zu ermöglichen, müssen Abhängigkeiten in beiden Systemen korrekt spezifiziert werden!

Fügen Sie Ihrem Projekt für die Implementierung des Rucksackproblems zunächst ein neues IntelliJ-Java-Modul *ga.problems* hinzu. Legen Sie in diesem Modul ein Paket *ga.problems.knapsack* an, welches später alle Klassen des Problems enthalten soll. Die im Folgenden genannten Klassen sollen jeweils die entsprechenden Interfaces bzw. abstrakten Klassen der vorgegebenen Schnittstelle implementieren.

2.1 Problem und Lösung [4 Punkte]

Implementieren Sie die Klassen *KnapsackProblem* und *KnapsackSolution*. Die *Problem*-Klasse soll generelle Informationen über das Rucksackproblem (z.B. Kapazität des Rucksacks) halten und über *createNewSolution()* neue *KnapsackSolutions* anlegen können. Beim Erstellen einer neuen Lösung soll dabei solange ein zufälliger Gegenstand in den Rucksack gepackt werden, bis keiner der noch übrigen Gegenstände mehr hinein passt. Für den Fall, dass alle im Problem spezifizierten Gegenstände zu schwer für den Rucksack sind, also kein einziger Gegenstand hineingepackt werden kann, soll eine *NoSolutionException* geworfen und die Optimierung abgebrochen werden.

Die *Solution*-Klasse soll lösungsspezifische Informationen halten (z.B. welche Gegenstände sich im Rucksack befinden). Gegebenenfalls macht es Sinn, zusätzliche (redundante) Informationen zu halten, um die Performanz des Programms zu verbessern.

Selbstverständlich können Sie auch weitere Klassen anlegen, falls Sie diese zur Modellierung von Problem und Lösung benötigen.

[6] https://de.wikipedia.org/wiki/Rucksackproblem

2.2 Fitness [2 Punkte]

Implementieren Sie die Klasse *KnapsackFitnessEvaluator*, welche die Fitness aller Lösungen einer Population berechnet. Nutzen Sie zur Berechnung der Fitness einer Lösung Java-Streams und Lambda-Ausdrücke!

2.3 Mutationsoperator [5 Punkte]

Implementieren Sie für das Rucksackproblem *einen* evolutionären Operator *KnapsackMutation*, der zufällig auf eine der folgenden Arten eine neue Lösung erzeugt:

- Ein zufälliger Gegenstand wird aus dem Rucksack entfernt.
- Aus den Gegenständen, die noch nicht gewählt wurden und noch in den Rucksack passen, wird ein zufälliger gewählt und in den Rucksack gelegt.

Wichtig ist dabei, dass hierbei die ursprüngliche Lösung nicht verändert wird. Veränderungen sollten also auf einer Kopie durchgeführt werden. Verwenden Sie zur Erzeugung von Kopien einer Lösung einen konkreten Copy-Konstruktor für *KnapsackSolutions*.

Achten Sie auch darauf, dass zur Erstellung einer neuen Lösung immer exakt *eine* der genannten Veränderungen durchgeführt wird. Sollte keine der beiden Varianten anwendbar sein, ist dies als Fehler (*EvolutionException*) zu behandeln und die Optimierung insgesamt abzubrechen.

Es soll nur *ein* evolutionärer Operator entstehen, der beide Varianten integriert. Verwenden Sie innere Klassen, um die unterschiedlichen Varianten der Mutation zu implementieren.

Hinweis

Die Teilaufgaben sind weitestgehend unabhängig voneinander lösbar. Für die Erstellung der Fitnessfunktion und des Mutationsoperators benötigt man jedoch zumindest eine minimale Implementierung (Stub) der *KnapsackSolution*.

3 Ausführung einer Optimierung [2 Punkte]

Erstellen Sie eine Klasse *ConcreteProblem*, die eine Optimierung anhand eines konkreten *KnapsackProblems* demonstriert. Das Beispielproblem soll dabei aus einem Rucksack mit einer Kapazität von 11 Gewichtseinheiten und zehn Gegenständen bestehen:

- `g1`(Gewicht:5, Wert:10)
- `g2`(Gewicht:4, Wert:8)
- `g3`(Gewicht:4, Wert:6)
- `g4`(Gewicht:4, Wert:4)
- `g5`(Gewicht:3, Wert:7)
- `g6`(Gewicht:3, Wert:4)
- `g7`(Gewicht:2, Wert:6)
- `g8`(Gewicht:2, Wert:3)
- `g9`(Gewicht:1, Wert:3)
- `g10`(Gewicht:1, Wert:1)

Initialisieren Sie in *ConcreteProblem* das Problem und führen Sie, mit einer Population der Größe 4, eine Optimierung über 10 Iterationen durch. Verwenden Sie dabei *TopK-Survival* und, sofern Sie diese implementiert haben, *TournamentSelection*. Geben Sie auf der Konsole für jede Lösung der finalen Population deren Fitness und die im Rucksack enthaltenen Gegenstände aus. Vergleichen Sie die Ergebnisse für unterschiedliche Werte des Parameters k des *TopKSurvival*.