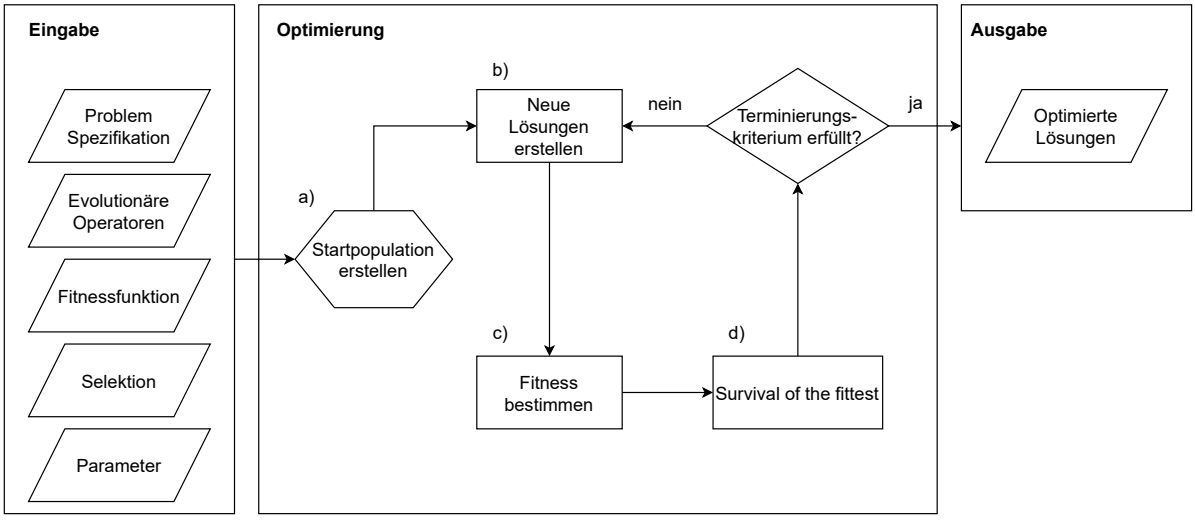

Abbildung 1: Schematische Darstellung des Ablaufs genetischer Algorithmen.

Vorstellung genetischer Algorithmen

In den nächsten zwei Tagen sollen Sie sich mit genetischen Algorithmen[1] beschäftigen. Genetische Algorithmen sind such-basierte heuristische Optimierungsverfahren. Sie werden in der Regel zur näherungsweisen Lösung von Problemen verwendet, für die eine exakte Lösung nur schwer oder unter hohem Zeitaufwand berechnet werden kann. Die Suche nach optimierten Lösungen ist dabei an die Konzepte natürlicher Evolution angelehnt.

Abbildung 1 stellt schematisch den Ablauf der durch einen genetischen Algorithmus durchgeführten Optimierung dar.

a) Genetische Algorithmen arbeiten üblicherweise auf einer *festgelegten Anzahl* möglicher Lösungen, der sogenannten Population. Zu einem gegebenen Problem wird daher zunächst eine Startpopulation (unoptimierter) Lösungen erstellt und die Qualität der Lösungen über eine sogenannte Fitnessfunktion bestimmt. Ausgehend von dieser Population wird iterativ nach besseren Lösungen gesucht.

b) Dazu werden, unter Verwendung evolutionärer Operatoren, aus den bestehenden Lösungen neue Lösungen abgeleitet. Dabei entsteht üblicherweise eine Menge von

[1] https://en.wikipedia.org/wiki/Genetic_algorithm

Lösungen, die sowohl alle Elemente der ursprünglichen Population, als auch die daraus abgeleiteten Nachkommen enthält.

Ein gängiger evolutionärer Operator ist die sogenannte *Mutation*. Bei dieser entsteht, analog zum Vorbild aus der Natur, ein Nachkomme durch eine kleine Veränderung einer bestehenden Lösung.

c) Nach Anwendung der evolutionären Operatoren wird über die Fitnessfunktion die Qualität aller Nachkommen ermittelt.

d) Ein Überlebensoperator wählt abschließend aus, welche Lösungen überleben und die Population der nächsten Iteration bilden. Hierbei wird meist das Prinzip des *Survival of the fittest* umgesetzt, d.h. qualitativ bessere Lösungen werden bevorzugt.

Die Schritte b), c) und d) werden solange zyklisch durchlaufen, bis ein vorher *festgelegtes Kriterium zur Termination* des genetischen Algorithmus (bspw. das Erreichen einer festgelegten Anzahl von Iterationen) erfüllt ist.

Vorgegebene Schnittstelle

In den folgenden Aufgaben sollen Sie zum einen ein Framework für genetische Algorithmen entwickeln und zum anderen ein Optimierungsproblem im Kontext dieses Frameworks modellieren, implementieren und durch Implementierung passender Operatoren lösen.

Ihre Implementierung soll dabei auf einer vorgegebenen Schnittstelle basieren, die als Java-Modul[2] im Sinne des Java Platform Module Systems (JPMS) implementiert wurde. Sie finden diese in Ilias[3] als IntelliJ-Projekt. Erstellen Sie zunächst selbst ein neues **leeres** Projekt (*File* → *New* → *Project* → *Empty Project*) *genetic-algorithm*. Importieren Sie (*File* → *Project Structure* → *Project Settings* → *Modules* → + → *Import Module*) anschließend in diesem das bereitgestellte Projekt als IntelliJ-Modul und verwenden Sie dieses Modul als Grundlage für die weiteren Arbeitsschritte.

Hilfe zum Umgang mit (JPMS)-Modulen in IntelliJ finden Sie in der Dokumentation von IntelliJ[4].

[2] https://www.oracle.com/corporate/features/understanding-java-9-modules.html
[3] https://ilias.uni-marburg.de/goto.php?target=file_2954857_download&client_id=UNIMR
[4] https://www.jetbrains.com/help/idea/creating-and-managing-modules.html#modules-idea-java

Durch die vorgegebene Schnittstelle lassen sich Aufgabe 1 und 2 unabhängig voneinander bearbeiten und lösen.

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