# Aufgabe 2: Rechenrätsel

Teilnahme-ID: 61743

Bearbeiter dieser Aufgabe:
Jonas Fennekohl

13. April 2022

1. Lösungsidee: 2
1.1 Erklären der gestellten Bedingungen: 2
1.3 Ausrechnen aller möglichen Kombinationen aus Operatoren: 3
1.4 Berechnen der Laufzeit: 5
1.5 Erweiterungsideen: 5
2. Umsetzung: 6
2.1 Wählen der Datenstruktur zum Abspeichern der Ergebnisse: 7
2.2 Erstellen aller möglichen Gleichungen: testRaetselOriginal 7
2.3 Erweiterung durch Klammern und Hoch: testRaetselOriginalMitKlammern 9
2.4 Ausrechnen der in 2.2 und 2.3 erstellen Gleichungen: evalEquation 10
2.5 Verbesserung der realen Laufzeit durch Multithreading: raetselErstellenMultithreading 12
2.5.1 Der Datentyp RaetselDatapack: 12
3. Beispiele: 13
3.1 Beispiele gemäß der Aufgabenstellung: 13
Länge = 2: 13
Länge = 3: 13
Länge = 8: 13
Länge = 12: 13
Länge = 14: 14
Länge = 15: 14
Länge = 18: 14
Länge = 19: 14
3.2 Beispiele gemäß meinen Erweiterungen: 14
Länge = 5: 14
Länge = 6: 14
Länge = 7: 15
Länge = 8: 15
4. Quellcode 15
4.1 Methode evalEquation: 15
4.2 Methode testRaetselOriginalMitKlammern; 17
4.3 Methode testRaetselOriginal: 18
Klasse RaetselDatapack: 19
4.4 Klasse MultithreadRaetselOriginal: 20

# 1. Lösungsidee:

## 1.1 Erklären der gestellten Bedingungen:

Als kleines Vorwort möchte in anmerken, dass ich die Aufgabenstellung so verstanden habe, dass alle genannten Bedingungen gleichzeitig erfüllt werden sollen. Außerdem werde ich in den mathematischen Notationen meiner Lösung Bruchstriche für „:“ und das Symbol „·“ für „*“ benutzen. Beispiel: 3:3 = $\frac{3}{3}$ , 3*4 = $3 \cdot 4$

Schauen wir uns zuerst an , welche Bedingungen gestellt werden und was mit diesen gemeint ist:

a) Das Rätsel eindeutig lösbar ist

Dies bedeutet, dass mit allen möglichen Kombinationen aus Operatoren bei gleichbleibenden Operanden das Ergebnis der Gleichung nur in dem gestellten Rätsel vorkommt. Beispiel:

$$3 \circ 4 \circ 3 \ = \ 15 \ \rightarrow \ 3 + 4 \cdot 3 \ = \ 15 \ \ oder \ \ 3 \cdot 4 + 3 = 15$$

Dadurch wird festgelegt, dass alle, laut den anderen Bedingungen, mögliche Kombinationen aus den Operatoren betrachten müssten, um sicherzustellen, dass Ergebnis einzigartig ist.

b) die Operanden einzelne Ziffern sind

Dies bedeutet: $Operanden \subseteq \{ -9, -8 \ldots 8,9 \}$. Da in der Aufgabenstellung nur positive Zahlen gezeigt werden kann man davon ausgehen, dass die Aufgabenstellung meint, dass nur positive ganze Zahlen im Intervall von [1…9] betrachtet werden sollen. Für die Lösung der Aufgabenstellung ist das aber erstmal unwichtig, weshalb in den Beispielen und in dem Programm zwar dazwischen unterschieden wird, es aber im Rest der Dokumentation keine Erwähnung mehr findet. Etwas weiteres was zwar nicht in der Aufgabenstellung angeführt wird ist, dass 0 & 1 in bestimmten Fällen, zu keinem relevanten Ergebnis führen.

c) Eine positive ganze Zahl als Ergebnis herauskommt

Dies bedeutet: $Ergebnisse \in \mathbb{Z}^+$ Trotz dieser Bedingung können weiterhin negative Operanden verwendet werden, ein Beispiel dafür wäre: $-3 \cdot -3 \ = 9$

d) Punkt- vor Strich-Rechnung angewandt wird

Dieser Punkt ist selbsterklärend. Wir haben zwei Klassen von Operatoren, Klasse 1 = {*,:}, Klasse 2 = {+,-}. In einer Gleichung müssen erst alle Operatoren der Klasse 1 verrechnet werden, bevor Klasse 2 beachtet wird.

e) Gleichrangige Operatoren linkassoziativ angewandt werden

Diese Bedingung fordert, dass Operatoren derselben Klasse von links nach rechts ausgerechnet werden und erst dann in die nächste Klasse übergegangen wird.

Die Bedingungen d), e) kombiniert ergeben Rechenregeln, welche durchgehend verwendet werden und jedem ein Begriff seien sollten.

f) Alle Zwischenergebnisse ganze Zahlen sind

Diese Bedingung ist mit b) verbunden. b) definiert, dass Operanden einzelne Ziffern sind, also durchgehend ganze Zahlen sind. Der einzige Operator, mit welchem Dezimalzahlen errechnet werden können ist also „:“(Teilen), da mit allen anderen Operatoren, wenn zwei ganze Zahlen miteinander verrechnet werden, immer eine ganze Zahl herauskommt. Das bedeutet dass in unserer Lösung, wenn geteilt wird, überprüft werden muss, ob das Ergebnis eine Dezimalzahl ist.

Wir können also zusammenfassend sagen, dass wir eine *Tiefe* des Rätsels haben, welche angibt, wie viele Operatoren in dem Rätsel verwendet werden sollen, eine Liste mit den möglichen Optionen *Operatoren* = {+,-,*,:}, eine Liste *Zahlen* mit den verwendeten Zahlen für das Rätsel, mit Tiefe+1 Elementen, welche aus der Liste $M\ddot{o}glicheOperanden = \{ -9, -8 \ldots 8,9 \}$ genommen werden.

Beispiel: Tiefe = 3; Zahlen = {5,9,3,-2}

Zum Berechnen aller Möglichkeiten wird die Gleichung so konstruiert, dass zwischen alle Zahlen Operatoren platziert werden. Dann wird die Gleichung ausgerechnet und alle anderen Bedingungen, die gestellt wurden auf sie angewendet(c), d), e), f)). Wenn ein valides Ergebnis erreicht wird, wird das Ergebnis in eine Liste geschrieben, aus welcher nachher ein Ergebnis und die dazu gehörige Gleichung genommen wird. Um die Bedingung a) zu erfüllen, wird geschaut, ob das Ergebnis genau einmal in der ganzen Liste vorhanden ist.

Die Idee ist es also einen Algorithmus zu entwickeln, welcher alle Möglichkeiten durchrechnet die Gleichung zu konstruieren und währenddessen die anderen Bedingungen beachtet. Außerdem soll er alle möglichen Gleichungen in einer Datenstruktur abspeichern und dabei Duplikate aussortieren. Genauso soll er sicherstellen, dass keine Dezimalzahlen vorkommen und die Operatoren/Rechenzeichen in der richtigen Reihenfolge anwendet.

## 1.3 Ausrechnen aller möglichen Kombinationen aus Operatoren:

Wir haben bisher festgestellt, dass wir aufgrund der Bedingung, dass die Gleichung einzigartig ist, wir alle möglichen Kombinationen aus Operatoren ausrechnen müssen, um sicherzustellen, dass das Rätsel eindeutig lösbar ist.

Um alle möglichen Kombinationen auszurechnen, können wir uns unsere Gleichung als eine Art Baum vorstellen. Dieser Baum sieht für die Zahlen = {3,7,2} zum Teil so aus.

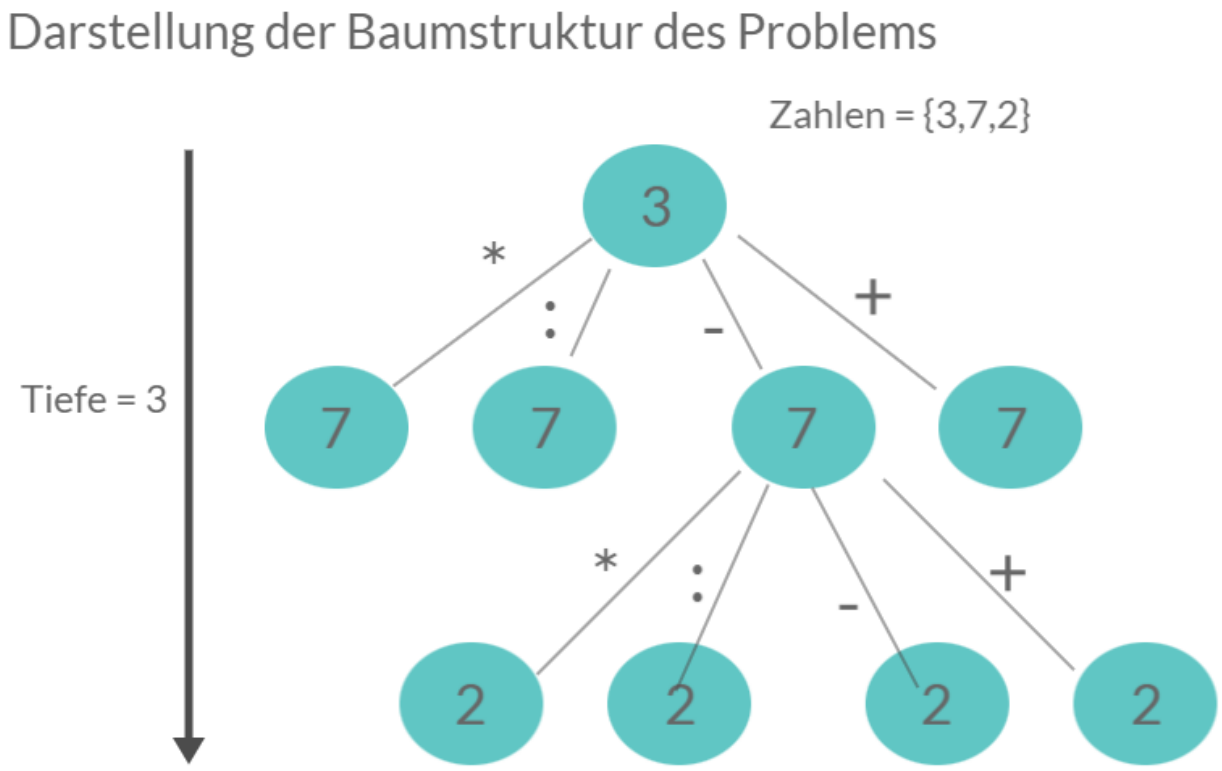


*Abbildung 1: Darstellung als Baum*

Wie wir in der Abbildung erkennen können, lässt sich für die Problemstellung ein Baumdiagramm zeichnen, welches alle mögliche Kombinationen der Operatoren abbildet. Von jedem Knoten gehen dabei 4 Äste ab, welche im Originalfall die 4 Operatoren repräsentieren. Die Tiefe des Baums ist dabei die Anzahl der unbekannten Operatoren. Mit dieser theoretischen Darstellung des Baums ist jeder Pfad in dem Baum eine mögliche Gleichung.

Da wir alle möglichen Pfade in dem Baum berechnen sollen, bietet sich an den Baum mithilfe einer Preorder-Traversierung zu traversieren. Dabei wird zuerst immer die Root ausgelesen und dann werden die Kinder von links nach rechts durchgegangen. Mit der Preorder-Traversierung erschafft man die verschiedenen Gleichungen und kann die erstellte Gleichung ausrechnen, sobald die Traversierung an einem Blatt angekommen ist. Um weniger Berechnungen durchführen zu müssen, können wir die bestehende Gleichung immer ausrechnen, falls der aktuelle Operator „:“ ist, auch wenn wir nicht an einem Blatt angekommen. Dies machen wir, da der Operator „:“erstmal der einzige Operator ist, welcher einen Teilbaum abtrennen kann, aufgrund der Punkt-vor-Strich Rechnung, was heißt das, wenn der Operator „:“ ein Dezimalergebnis ergibt, bleibt dieses durchgehend als Zwischenergebnis bestehend, weshalb der Teilbaum entfernt werden kann und folglich der Rest auch nicht mehr ausgerechnet werden muss. So muss man statistisch gesehen weniger rechnen, da so erstmal 25% mehr Rechnungen ausgeführt werden müssen, da insgesamt die Wahrscheinlichkeit das beim Teilen ein Zwischenergebnis mit Dezimalstellen rauskommt größer als 25% ist, da nur bei den Rechnungen: $\frac{GeradeZahl1}{GeradeZahl2}, 2 < 1$, teilen durch 1 etc. ein Ergebnis rauskommt, welches benutzt werden kann. Das bedeutet das es sich rechnerisch lohnt : immer auszurechnen, wenn wir die Äste danach abbrechen können.

## 1.4 Berechnen der Laufzeit:

Wir haben hier einen Brut Force Ansatz, das bedeutet, dass alle möglichen Varianten ausgerechnet werden müssen. Das ist bei dieser Aufgabenstellung leider nötig, da sonst nicht garantiert werden kann, dass das Ergebnis wirklich einzigartig ist. Deshalb ergibt sich für den schlechtesten Fall eine Laufzeit von O($n^a$). n ist hierbei die Anzahl der unbekannten Operatoren, die verwendet werden sollen, also die Länge des Rätsels, a ist die Anzahl der verschiedenen Operatoren. Der schlimmste Fall heißt das kein Teil des Baums entfernt werden kann.

Die bestmögliche Laufzeit, für die originale Aufgabenstellung ist O($n^{a-1}$), diese tritt ein, wenn immer, wenn geteilt wird, eine Dezimalzahl herauskommt und deshalb nie die Teilbäume des „:“ Operators ausgerechnet werden müssen.

Die durchschnittliche Laufzeit wäre O($b \cdot n^a$), wobei b der Faktor wäre, der angibt, wie viele der Teilbäume im Durchschnitt mit zufälligen Operanden wegfallen durch den „:“ Operator und wie viel von dem Baum mit $n^a$ Ästen bleibt. Da Vorfaktoren wegfallen, ist die durchschnittliche Laufzeit auch O($n^a$).
Im Originalbeispiel ist a=4 und n variierend.

## 1.5 Erweiterungsideen:

Da wir hier Rechenrätsel stellen sollen ist die einfachste und logischste Erweiterung diese Rechenrätsel zu erweitern. Dafür haben wir verschiedene Möglichkeiten:

1. Andere Rechenzeichen/Operatoren einführen z.B. „^“(hoch), „(“ und „)“ um die Art und Weise zu verändern, wie die Rätsel gestellt werden. $3^4 \rightarrow 3^{4-8*3}$. Die Rätsel werden komplizierter und deutlich vielfältiger durch die Einführung von Klammern, da nun z.B. ein Term unter dem Bruchstrich etc. stehen kann.

2. Die zweite Art ein Rechenrätsel zu verändern wäre, die Zahlen, die verrechnet werden zu verändern und dabei z.B. Dezimalzahlen zu erlauben. Dieser Erweiterung würde vor allem den Schwierigkeitsgrad der Rätsel in einer Art und Weise erhöhen, ohne wirklich neue Aspekte des Rätsels zu bringen, da die bekannten Teile verändert werden und nicht die Unbekannten. So könnte man über Dezimalzahlen verschiedene Schwierigkeitsgrade einführen. Level 1 = bis zu einer Dezimalstelle, Level 5 = bis zu fünf Dezimalstellen. … Damit würde aber das Rätsel nicht wirklich erweitert werden, sondern nur schwieriger gemacht. Da eine Erweiterung der möglichen Zahlen auf Dezimalzahlen, die Bedingung f) fast redundant macht, da diese der Erweiterung im Weg steht, könnte man auch

3. Limitierende Bedingungen, wie c), b) und f) auflösen, um mehr Varianten zu schaffen, in denen die Rätsel gestellt werden. Vor allem für den „:“ Operator ist f) tödlich, da in den meisten Kombinationen von Zahlen eine Dezimalzahl als Ergebnis herauskommt. Diese Bedingung sorgt für weniger Varianz in dem Rätsel, weshalb entfernen der Bedingung f) die Rätsel logisch erweitert.

Nachdem wir die drei Richtungen der Erweiterungen aufgezählt haben, schauen wir uns an, wie diese unseren „Zielsatz“ aus 1.1 ändern müssen, um die Erweiterungen aufzunehmen.

Auswirkungen der Erweiterungen zusammengefasst:

1. Wir können also zusammenfassend sagen, dass wir eine *Tiefe* des Rätsels haben, welche angibt, wie viele Operatoren in dem Rätsel verwendet werden sollen, eine Liste mit den möglichen Optionen *Operatoren* = {+,-,*,:,^,(,)}, eine Liste *Zahlen* mit den verwendeten Zahlen für das Rätsel, mit Tiefe+1 Elementen, welche aus der Liste $MöglicheOperanden = \{ -9, -8 \ldots 8,9 \}$ genommen werden. Beispiel: Tiefe = 3; Zahlen = {5,9,3,-2}

   Zum Berechnen aller Möglichkeiten wird die Gleichung so konstruiert, dass zwischen alle Zahlen Operatoren platziert werden. Dabei muss darauf geachtet werden, dass Klammern, welche geöffnet werden, auch geschlossen werden. Außerdem sollte in der Gleichung eine Rechnung stehen(Gewünscht: 3*(4-2) = 6 und nicht: 3*(4)-2 = 10), um redundante Berechnungen zu vermeiden. Dann wird die Gleichung ausgerechnet und alle anderen Bedingungen, die gestellt wurden auf sie angewendet(c), d), e), f)). Wenn ein valides Ergebnis erreicht wird, wird das Ergebnis in eine Liste geschrieben, aus welcher nachher ein Ergebnis und die dazu gehörige Gleichung genommen wird. Um die Bedingung a) zu erfüllen, wird geschaut, ob das Ergebnis genau einmal in der ganzen Liste vorhanden ist.

   Unser Baum hat damit 3 mehr Operanden, für denen es Äste geben muss. Dabei gibt es aber neue Regeln. Die letzte offene Klammer wird nur bis der Tiefe-2 geöffnet, da die Klammer wieder geschlossen werden muss. Wenn man eine Klammer öffnet und im nächsten Operator die Klammern wieder schließt, ist die Klammer redundant und unnötig, da keine Berechnung in der Klammer stattfindet. Außerdem kann eine Klammer nur geschlossen werden, wenn noch eine offene Klammer besteht und mindestens schon ein Operator zwischen der Klammer und der zu schließenden Klammer steht. Dabei ändert sich der Wert von *a = 4* bei der Originalaufgabe zu *a = 7* durch die neuen Operatoren

2. Wir können also zusammenfassend sagen, dass wir eine *Tiefe* des Rätsels haben, welche angibt, wie viele Operatoren in dem Rätsel verwendet werden sollen, eine Liste mit den möglichen Optionen *Operatoren* = {+,-,*,:}, eine Liste *Zahlen* mit den verwendeten Zahlen für das Rätsel, mit Tiefe+1 Elementen, welche aus der Liste $MöglicheOperanden = \{\mathbb{Q} \mid -10 > x < 10\}$ oder $MöglicheOperanden = \{\mathbb{Q}\}$ genommen werden, je nachdem welche Zahlenmenge genommen wird. Die Frage dabei ist, ob nur Dezimalzahlen eingeführt werden, oder auch Zahlen <= -10 oder >=10 verwendet werden. Wenn Dezimalzahlen eingeführt werden, dann werden auch Teilbäume weitergerechnet, in denen Dezimalzahlen auftreten.

3. Die Gleichungen/Ergebnisse werden auf weniger Bedingungen überprüft. Außerdem werden, wenn man die Bedingung f) entfernt die Teilbäume, welche Dezimalzahlen durch Division beinhalten doch errechnet.

## 2. Umsetzung:

Meine Lösung habe ich in der Programmiersprache Java geschrieben.

## 2.1 Wählen der Datenstruktur zum Abspeichern der Ergebnisse:

Wenn wir uns meine Lösungsidee anschauen, sehen wir das wir eine Datenstruktur brauchen, in welcher unsere errechneten Ergebnisse zwischenspeichern können. Dafür habe ich eine HashMap gewählt, da wir zwei Werte haben, die wir abspeichern müssen, dass Ergebnis der Gleichung und die dazugehörige Gleichung. Eine HashMap hat dabei Key-Value Pairs, was bedeutet, dass man beim Hinzufügen in die Datenstruktur einem Wert einen Schlüssel zuordnet, über welchen dieser Wert erreicht werden kann. Diese Logik bietet sich für unsere Situation an, da wir das Ergebnis als Schlüssel benutzen können, um die Gleichung als Wert abzuspeichern.

Eine HashMap funktioniert ähnlich wie ein Array, mit dem Unterschied, dass man keinen Index hat, sondern ein Key. Man hat aber trotzdem einen direct-access zu dem Wert, ohne das gesucht werden muss, wenn man den Wert des Keys abfragt.

Dies wird wichtig, wenn wir betrachten, dass wir doppelte Werte aussortieren müssen, da wir so direkt nachschauen können, ob ein errechneter Wert bereits in der Liste vorhanden ist. So wird die Suche gespart, die bei einem normalen Array nötig wäre, O(log(n)) gegen O(1). Um diesen doppelten Wert abzuspeichern, damit auch wenn ein Ergebnis mehr als mal errechnet wird, trotzdem aussortiert wird, benutzte ich eine ArrayList. Dies ist nötig, da sonst, wenn ein Ergebnis h ($h \in \{\mathbb{Z} \mid ungerade\ Zahl\ > 2\}$) mal errechnet wird, es am Ende trotz mehrfachen Vorkommens beim Rechenprozess in der HashMap zu finden ist, da es erst hinzugefügt wird, dann wieder entfernt wieder etc. So wird jedes entfernte Ergebnis in der ArrayList abgespeichert, solange es auch dort noch nicht vorhanden ist. Um den Speicherplatz zu optimieren, wird nur das Ergebnis abgespeichert, da sich an diesem in der Aufgabenstellung orientiert wird.

## 2.2 Erstellen aller möglichen Gleichungen: testRaetselOriginal

Um zu verstehen, wie mein Algorithmus die möglichen Gleichungen(alle Kombinationen der Rechenzeichen/Operatoren) erstellt, müssen wir uns erstmal die Visualisierung in 1.3 noch einmal anschauen.

Da wir die theoretische Darstellung des Baums nur zu dem Verstehen der Logik, wie man die einzelnen Gleichungen errechnet, brauchen wird in meinem Programm der Baum nicht als Datenstruktur erstellt, sondern nur theoretisch errechnet.

Da wir alle möglichen Pfade in dem Baum berechnen sollen, bietet sich eine rekursive Programmierung an. Hier wurde in 1.3 von einer Preorder-Traversierung gesprochen, also zuerst die Root und dann die Children durchzugehen. Meine Methode erstellt die Gleichung nach der Logik.

Die Methode ist rekursiv, das heißt sie ruft sich mehrfach selbst auf. Außerdem gibt sie einen Boolean zurück, welcher anzeigt, ob das Blatt erfolgreich war. Diese HashMap ist nach dem Schema aufgebaut, welches auch in 2.1 präsentiert ist. Der Methode werden außerdem die Parameter int currTiefe, int maxTiefe, String alteGleichung, String operator weitergegeben

Da der Algorithmus, den in ich 1 entwickelt habe als ein Baum modelliert werden kann, bei welchem die Tiefe des Baums der relevante Faktor für die Abbruchbedingung ist. Die Tiefe des Baums ist dann die Anzahl der Operatoren, die verwendet werden.

Alle Ergebnisse werden in der HashMap Gleichungen abgespeichert.

Zuerst wird die Abbruchbedingung, ob die Methode gerade an einem Blatt, also an der maximalen Tiefe(gewünschte Anzahl der Operatoren) angekommen ist, abgefragt. Wenn ja wird die finale Gleichung am Blatt als String erstellt, indem man der alten Gleichung(alteGleichung) den weitergegebenen Operator(operator) hinzufügt. Außerdem wird die Zahl aus dem Array zahlen abgerufen, welche an der aktuellen Tiefe(currTiefe) steht. Die aktuelle Tiefe wird dabei als Index verwendet. In dem Array zahlen werden die Zahlen gespeichert, welche für die Gleichung verwendet werden. Diese werden am Anfang des Programms zufällig generiert in dem Bereich, welcher angegeben wurde. Die Länge des Arrays ist gleich der Tiefe des Baums. Da man für das Rätsel aber eine Zahl mehr als Operatoren braucht, wird beim Aufruf der Methode testRaetselOriginal eine zusätzliche Zahl generiert und als alteGleichung übergeben, welche nicht in dem Array vorhanden ist. Diese Zahl ist praktisch gesehen die Wurzel des Baums. Der erstellte String wird dann als currGleichung gespeichert.

Beispiel: alteGleichung = „3*4-2“, operator =“:“, zahlen[currTiefe] = 7, = Gleichung + Operator + Zahlen[aktuelle Tiefe] = 3*4-2/7

Nachdem die Gleichung als String erstellt wurde wird die Gleichung danach mit der Methode evalEquation(Siehe 2.3) ausgerechnet/evaluiert. Danach wird kontrolliert, ob das Ergebnis der Gleichung bereits vorhanden ist. Außerdem kontrolliert die Methode auch, ob ein Ergebnis überhaupt errechnet wurde, indem überprüft wird, ob das Ergebnis != null ist. Um sicherzustellen, dass das Ergebnis bisher noch nicht errechnet wurde, wird geschaut, ob das Ergebnis in der Liste entfernteErgebnisse vorhanden ist. In dieser Liste werden alle errechneten Ergebnisse eingefügt, damit auch Teilbaum übergreifend kontrolliert werden kann, ob das Ergebnis doppelt vorkommt. Wenn man nur in der Methode selbst guckt, würde sonst nur überprüft werden, ob das Ergebnis bereits in dem Teilbaum am Blatt vorhanden ist. Dies ist so, da die Methode rekursiv bis zum Blatt aufgerufen wird, dann die Ergebnisse berechnet und in dem Methodenaufruf im Blatt überprüft, ob es schon in der HashMap vorhanden ist, weshalb es nochmal eine externe Liste geben muss, die die Ergebnisse sammelt.

Wenn das Ergebnis nicht in der Liste vorhanden ist, dann wird es zur HashMap Gleichungen hinzugefügt. Außerdem wird das Ergebnis auch in die Liste selbst hinzugefügt. Wenn es schon in der Liste vorhanden ist, wird es in keine der beiden Datenstrukturen hinzugefügt, um redundante Einträge in entfernteErgebnisse zu vermeiden.

Falls die Methode nicht an einem Blatt angekommen ist, wird die Gleichung nach demselben Schema erweitert. Wenn der aktuell hinzugefügte Operator „:“ ist, wird die Gleichung auch ausgerechnet, wenn sie nicht an einem Blatt ist, um die Bedingung f) möglichst früh zu bemerken und damit unnötige Rechnungen zu sparen, welche durch Teilung mit Dezimalergebnissen entstehen. Wenn entweder der Operator nicht „:“ war oder ein valides Ergebnis für „:“ errechnet wurde, geht die Methode zum rekursiven Teil über. Dabei werden mit einer for-Schleife die ArrayList rechenzeichenOriginal durchgegangen. Die Liste enthält die Operatoren {“+“.“-“.“*“.“:“}. Dadurch werden alle möglichen Kombinationen ausgerechnet, da jeder Operator weitergegeben wird. Die Methode ruft sich dann selbst auf. Der rekursive Methodenaufruf sieht dann so aus:

```
testRaetselOriginal(currTiefe, maxTiefe, currEquation,
rechenzeichenOriginal[i]
```

Dabei werden die aktualisierten Werte für currTiefe, currEquation und rechenzeichenOriginal weitergegeben. Am Ende der Methode wird true zurückgegeben, um anzuzeigen, dass die Methode richtig ausgeführt wurde.

## 2.3 Erweiterung durch Klammern und Hoch: testRaetselOriginalMitKlammern

Als Erweiterung habe ich mich entschieden mich auf das Hinzufügen von Operatoren, vor allem auf die Klammern zu fokussieren, da dies die anspruchsvollste Erweiterung war. Bei den anderen Operatoren hätte ich nur das zufällige Generieren der Zahlen verändern müssen und if() Abfragen in evalEquation etc. löschen/ändern müssen. Zum Einführen von dem Hoch-Zeichen „^“ muss man nur diese Zeichen in die Liste der möglichen Rechenzeichen aufnehmen. Die Klammern sind deutlich komplizierter.

Die Methode testRaetselOriginalMitKlammern funktioniert fast genau gleich wie testRaetselOriginal, mit ein paar Unterschieden. Wird das Array String[] rechenzeichenErweitert verwendet. In diesem Array sind die Rechenzeichen {„+“, „-“, „*“, „:“, „^“} gespeichert. Für diese werden dann die Kombinationen nach dem alten Prinzip generiert. Durch die Erweiterung ergeben sich auch neue Klassen, an die sich unsere .evalEquation Methode halten muss. Klasse 1 = {„(“, „)“}, 2 = {„^“}, 3 = {„*“, „:“}, 4 = {„+“, „-“}. Diese Klassen stellen weiterhin nur die Reihenfolge dar, wie die Operatoren ausgerechnet werden müssen.

Das Schwierige an der Erweiterung ist das Implementieren der Logik zum Generieren von Klammern. Jede Klammer, die geöffnet wird, muss auch wieder geschlossen werden. Um zu tracken, wie viele Klammern noch geschlossen werden müssen, hat diese Methode einen zusätzlichen Parameter AnzahlOffeneKlammern. Dieser hält fest, wie viele Klammern noch offen sind. Generell kann man für die Verwendung von Klammern folgende Bedingungen festhalten:

1. Jede Klammer, die geöffnet wird, muss auch wieder geschlossen werden.
2. Zwischen jeder Klammer die geöffnet und geschlossen wird, muss mindestens eine Berechnung durchgeführt werden, da die Klammer sonst redundant wäre. Beispiel: Richtig: 6(3+4):2+8, Falsch: 6(3)4:2+8
   Das bedeutet auch das die letzte Klammer maximal 3 Operatoren vor Ende geöffnet werden darf.
3. Wenn vor der Klammer kein Rechenzeichen steht, wird die Klammer auch nach den normalen Rechenregeln als Multiplizieren gewertet. (Muss in evalEquation eingebaut werden.)

Die Funktion testRaetselOriginalMitKlammern muss also die Bedingungen 1, 2 erfüllen. Um die Bedingung 1 zu erfüllen, gibt es den Parameter AnzahlOffeneKlammern, welcher festhält, wie viele offene Klammern die Gleichung hat. Nur wenn eine Gleichung an einem Blatt 0 offene Klammern hat, wird sie als valide Gleichung betrachtet und zur HashMap hinzugefügt.

Die zweite Bedingung wird durch verschiedene if-Abfrage abgedeckt. Wir haben dabei jeweils einen Fall in dem keine Klammer geöffnet werden darf und zwei Fälle, in denen keine Klammer geschlossen werden darf.

Es darf keine Klammer geöffnet werden, wenn die Anzahl der restlichen Operatoren(currTiefe-maxTiefe)<= die Anzahl der offenen Klammern(AnzahlOffeneKlammern)ist. Denn wenn dort eine Klammer geöffnet wird, dann können nicht mehr alle Klammern geschlossen werden, weshalb die entstehende Gleichung ungültig wäre. Wenn diese Bedingungen erfüllt sind, berechnet die Methode eine Variante, in der die Klammer an der Stelle geöffnet wird. Dafür wird eine offene Klammer als operator dem Methodenaufruf übergeben

Es darf keine Klammern geschlossen werden, wenn der letzte Operator eine offene Klammer ist, da dies bedeutet, dass sich redundante Klammern ergeben. Es darf auch keine Klammer geschlossen werden, wenn keine offenen Klammern mehr bestehen. Wenn diese Bedingung erfüllt ist berechnet die Methode eine Variante, wo eine Klammer dort geschlossen wird. Dafür wird eine geschlossene Klammer als operator dem Methodenaufruf übergeben

Wie die andere Bedingung erfüllt wird, schauen wir uns jetzt in 2.4 an.

## 2.4 Ausrechnen der in 2.2 und 2.3 erstellen Gleichungen: evalEquation

Um die erstellen Gleichungen auszurechnen, musste ich eine eigene Methode .evalEquation(String equ) schreiben, welche die Gleichungen unter Berücksichtigung der Bedingungen ausrechnet und dabei ein Double als Ergebnis der Gleichung zurückgibt, da es in Java keine eigene Möglichkeit gibt eine Gleichung, welche sich verändert im Code selbst zu verändern oder durch eine Methode zu evaluieren. Dabei wird der String zuerst in eine ArrayList<String> konvertiert, um das Ausrechnen einfacher zu machen. Dabei wird der werden die einzelnen Zahlen und Operatoren des Strings zerteilt. Beispiel: „3-3*4-3“→ $\{3, -, 3, *, 4, -, 3\}$

Außerdem wird in der Methode durchgehend mit Double gerechnet, um möglicherweise mit Dezimalzahlen zu rechnen, mit dem Ziel durchgehende Datentypen zu verwenden, da andere Algorithmen die die .evalEquation Funktion benutzen, für die Erweiterungen Double verwenden müssen. Es wird außerdem die Wrapperklasse für double verwendet, um auch null, als ungültiges Ergebnis zurückgeben zu können.

Wenn „(“ und „)“ in der Liste vorhanden sind, wird die Liste mit einer for-Schleife durchgegangen, bis die erste Klammer auf „(“ gefunden wird. Dann wird eine neue for-Schleife gestartet, die ab dem Index die Liste weitergeht und dabei notiert, wie viele geöffnete und geschlossene Klammern gefunden wurden. Wenn die for-Schleife startet, wurde bereits eine offene Klammer gefunden, weshalb der Wert anzahlKlammerAuf = 1 ist und AnzahlKlammerZu = 0. Die Schleife läuft so lange, bis anzahlKlammerAuf = AnzahlKlammerZu ist. Denn wenn dies eintritt, hat man ein die zusammengehörigen Klammern gefunden. Beispiel: 3(4-5(8*2+3)8:2). Wenn man von der ersten offenen Klammer bis zur nächsten geschlossenen gehen würde, wäre die Gleichung in der Klammer „4-5(8*2+3“, welches ein ungültiges Ergebnis ist. Deshalb filtert mein Algorithmus zuerst die Gleichung in der ersten Klammer raus „4-5(8*2+3)8/2“. Der richtige, also der letztere Term wird dann in einem rekursiven Methodenaufruf von evalEquation notfalls weiter vereinfacht und dann ausgerechnet. Der Rückgabewert daraus, -376, wird dann in die Ausgangsgleichung eingesetzt:

3(4-5(8*2+3)8:2 → 3*-376*8:2. Dann wird der Rest der Gleichung von der Funktion ausgerechnet. Genauso funktioniert die Zählmechanik, auch wenn man voneinander getrennte Klammern hat: 4-3(5+6)9-8(4+6), erster Term „(5+6)“, zweiter Term „(4+6)“. Der zweite Term wird auch gefunden, da die Methode, sobald sie ein Klammerpaar gefunden und aufgelöst hat, wieder in die originale for-Schleife zurückgeht und den Rest der ArrayList auch auf weitere Klammern untersucht. Sobald die beiden Gleichungen gefunden worden sind, wurden sie nacheinander rekursiv gelöst.

Wenn nach dem Durchgehen der Liste immer noch Klammern vorhanden sind, heißt es, dass eine(oder mehr) offene/geschlossene Klammern zu viele sind und die Gleichung deshalb ungültig ist. Dann gibt die Methode null zurück, um ein ungültiges Ergebnis anzuzeigen. Dies sollte bei meinen generierten Gleichungen nicht vorkommen, dient aber als Vorsichtsmaßnahme.

Solange „^“ in der Liste vorhanden ist, wird wieder in der Liste nach „^“ gesucht. Wenn das Zeichen an der Stelle index gefunden wurde, werden die Werte bei index-1 und index+1 mit der Math.pow() Methode aus der Math Bibliothek von Java verrechnet. Und der Teil der Liste von index-1 bis index+1 mit dem Ergebnis ersetzt. Dann wird zur nächsten Klasse der Operatoren übergegangen.

Solange entweder „*“ oder „:“ in der ArrayList vorhanden sind, wird die Liste von Anfang zum Ende durchgegangen und wenn einer der beiden Operatoren an der Stellen index gefunden wird, werden die Werte an den Stellen von index-1 und index+1 mit dem jeweiligen Operator verrechnet. Falls der Operator „:“ ist wird außerdem geguckt, ob bei der Berechnung Dezimalstellen entstanden sind. Dafür wird die Abfrage „`if(temp%1 != 0)`“ benutzt. temp ist dabei das Ergebnis der Rechnung. In Java berechnet man mit dem Operator % das Modulo der Zahl, also den Rest, der beim Teilen mit der Zahl entsteht. Wir teilen hier das $\frac{Ergebnis}{1}$, um einen Rest, also Dezimalstellen feststellen zu können. Wenn ein Rest besteht, gibt die Methode null zurück, um ein ungültiges Ergebnis zu symbolisieren.

Wenn alle Operatoren der Klasse 3(„*“, „:“) ausgerechnet wurden, geht die Methode zu denen der Klasse 4(„+“, „-“) über. Dann wird der Prozess wiederholt, solange diese Operatoren noch existieren, geht eine for-Schleife die Liste durch und sucht nach den Operatoren. Wenn diese dann an einem Index gefunden werden, werden die Werte bei index-1 und index+1, wie gewohnt mit dem Operator verrechnet.

Nachdem dann keine der (bekannten) Operatoren mehr vorhanden sind kontrolliert die Funktion, ob die Länge der Liste, welche die Gleichung darstellt, gleich 1 ist. Wenn sie gleich 1 ist, heißt es, dass ein Ergebnis errechnet wurde und die Stellung der Funktion nicht fehlerhaft war. Wenn dem so ist gibt die Funktion das errechnete Ergebnis zurück, sonst wird null zurückgegeben, um einen Fehler anzuzeigen. Eigentlich sollte es in dem Kontext, wo die Funktion in dem Programm genutzt wird, nicht dazu kommen, da sämtliche Eingaben vom Computer, ohne Außeneinfluss, generiert sind und deshalb eigentlich keinen Fehler enthalten. Diese Version meiner .evalEquation Funktion geht iterativ für die Klassen 2-4 und rekursiv für die Klammern in Klasse 1 vor. Außerdem hat die Funktion eine Laufzeit von O(n), da jedes Element der List je einmal durchgegangen wird.

## 2.5 Verbesserung der realen Laufzeit durch Multithreading: raetselErstellenMultithreading

Da wir mit einer theoretischen Laufzeit von O($a^n$) arbeiten, skaliert der Algorithmus zum Erstellen der Rätsel sehr schnell in lange reale Laufzeiten, um Rätsel von ca. 14 Stellen aufwärts zu generieren(ca. 11 min mit dem in 2.2 vorgestellten Ansatz). Das liegt an der exponentiell skalierenden Laufzeit. Um die ganze Generierung zu Beschleunigen kann man in Java etwas namens Multithreading benutzen. Multithreading in Java ist notwendig, weil Java standartmäßig nur einen Rechencore benutzt. Das bedeutet, dass Java nur einen meist kleinen Teil der CPU belastet, weshalb Java Programme normalerweise wenig CPU-Leistung benötigen. Da wir hier aber unsere schlecht skalierende Komponente nicht die Rechenleistung, sondern die Zeit ist, habe ich Multithreading verwendet, um mehrere Cores zum Rechnen zu benutzen.

In Java gibt es mehrere Möglichkeiten Multithreading zu betreiben. Man kann mit einzelnen Threads oder Runnables arbeiten. Diese bietet sich für die Situation nicht an, da man die Ergebnisse der Rechnungen nachher braucht, um zu kontrollieren, dass kein Ergebnis doppelt vorkommt.

Deshalb habe ich mich in der Art und Weise, wie neue Threads gestartet werden für einen neueren Ansatz in Java entschieden, dem Callable Interface. Das Callable Interface gibt eine Methode vor, die in der Klasse implementiert werden muss, die Call Methode. In der Call Methode kann man im Vergleich zur Run Methode vom Runnable auch Parameter verwenden und ein Objekt zurückgeben.

Die Klasse, die ich zum Multithreading programmiert habe, heißt MultithreadRaetselOriginal. Sie implementiert das Callable Interface mit dem Datentyp RaetselDatapack. Die Klasse RaetselDatapack ist das Objekt was die Call Methode zurückgeben wird. Sie wird dazu benutzt, die Daten die errechnet geworden sind zu speichern. Auf die Funktion der Klasse gehe ich später noch genauer ein.

In die Klasse wurden die testRaetselOriginal, evalEquation und alle Methoden und Objekte, die damit zusammenhängen kopiert. Die Methode call() enthält eine veränderte Version des Codes von testRaetselOriginal. Der Code wurde so weit verändert, dass in dem neuen Datentyp RaetselDatapack alle Daten gespeichert und verwaltet werden und das begrenzt viele neue Threads gestartet werden durch rekursiven Aufruf eines Objekts. Das passiert, bis man mit dem zweiten Operator fertig ist. Er werden nicht durchgängig neue Threads gestartet, denn der Prozess die Threads zu verwalten und nachher auszulesen, benötigt so viel Zeit, das dies relevant werden würde. Außerdem bremsen sich Threads gegenseitig aus, weshalb ich mein Programm auf eine Limitation von 16 gestellt habe, da mein Prozessor 16 virtuelle Cores hat. So nutzt das Programm diese optimal, ohne dass sich Threads gegenseitig ausbremsen.

### 2.5.1 Der Datentyp RaetselDatapack:

Die Klasse RaetselDatapack ist dazu gedacht, die errechneten Daten so zu verwalten, wie es in der normalen Version(2.2) die HashMap Gleichungen und die ArrayList errechneteErgebnisse getan haben. Aus dem Grund hat die Klasse auch eine ArrayList<Double> ergebnisse und eine HashMap<Double, String> gleichungen. Die Klasse dient als Wrapperklasse um beide Datenstrukturen zurückgeben zu können. Die Klasse hat außerdem die Methode merge, welche RaetselDatapack data1, data2 als Parameter hat und ein RaetselDatapack Objekt zurückgibt. Sie ist

dazu da, zwei Datapacks zu kombinieren und die Werte der einzelnen auf Dopplungen zu überprüfen und in ein Objekt zusammenzufassen.

Kommen wir zurück zur Callable. In der Klasse gibt es ein RaetselDatapack Objekt data, welche die Daten der Klasse abspeichert. In diesem Objekt werden die Ergebnisse und Gleichungen gespeichert und das Objekt wird am Ende der Call Methode zurückgegeben.

Die Callabels werden mithilfe von einem ExecutorSercive gestartet, welcher die Callables in eine ArrayList<Future<RaetselDatapack>> ablegt, wonach gewartet wird, bis alle Threads fertig sind. Dann werden die Futures ausgelesen und alle Datapacks mit .merge kombiniert, bis am Ende aus dem fertigen Packet ein zufälliges Ergebnis ausgewählt wird und als Rätsel ausgegeben wird.

## 3. Beispiele:

Bei mir werden im Programm ? als Symbole für unbekannte Operatoren benutzt und nicht ◦.

### 3.1 Beispiele gemäß der Aufgabenstellung:

#### Länge = 2:

Rätsel: -3?-3=0.0; Lösung: -3--3=0.0

Rätsel: 2?9=11.0; Lösung: 2+9=11.0

Rätsel: -8?-9=72.0; Lösung: -8*-9=72.0

Rätsel: 4?3=12.0; Lösung: 4*3 = 12

#### Länge = 3:

Rätsel: 7?8?3=59.0; Lösung: 7*8+3=59.0

Rätsel: -4?7?-6=168.0; Lösung: -4*7*-6=168.0

Rätsel: 3?9?-9=84.0; Lösung: 3-9*-9=84.0

Rätsel: -2?-4?-4=14.0; Lösung: -2+-4*-4=14.0

#### Länge = 8:

Rätsel: -6?-1?-4?-8?5?-1?-1?7=150.0; Lösung: -6--1+-4*-8*5--1--1+7=150.0

Rätsel: 7?-7?-5?2?-5?4?-6?3=496.0 ; Lösung: 7*-7*-5*2--5+4+-6+3=496.0

Rätsel: 3?-6?6?7?6?-6?3?3=720.0; Lösung: 3+-6*6-7*6*-6*3+3=720.0

Rätsel: -7?9?-3?6?5?9?1?-2=117.0 ; Lösung: -7*9:-3+6-5*9*1*-2=117.0

#### Länge = 12:

Rätsel: 5?3?7?6?-2?-8?-4?3?8?5?-1?1=3944.0; Lösung: 5*3*7+6:-2+-8*-4*3*8*5--1+1=3944.0

Rätsel: 7?6?7?-3?4?9?-8?-2?-5?-7?3?-9=1469.0; Lösung: 7-6*7*-3*4+9+-8:-2--5*-7*3*-9=1469.0

Rätsel: -8?-5?9?8?-2?6?-3?9?3?4?-3?-8=31475.0; Lösung: -8*-5*9+8*-2*6*-3*9*3*4--3+-8=31475.0

Rätsel: -2?-4?-8?-3?-3?7?3?7?-3?-9?8?-2=1686.0; Lösung: -2*-4--8*-3*-3*7:3+7*-3*-9*8+-2=1686.0

### Länge = 14:

Rätsel: 9?7?1?-1?4?8?6?-8?-5?8?-3?-5?6?1?-4=5686.0; Lösung: 9+7-1--1+4*8-6*-8*-5*8*-3--5*6*1*-4=5686.0

Rätsel: 7?5?9?-8?5?1?3?-3?2?-4?7?3?-4?-6?1=241889.0; Lösung: 7-5*9+-8*5*1*3*-3*2*-4*7*3*-4--6+1=241889.0

Rätsel: 9?-2?3?3?-4?3?9?9?3?-2?1?7?6?-7?8=14285.0; Lösung: -9*-2*3*3--4*3-9:9-3*-2*1*7*6*-7*8=14285.0

Rätsel: -3?1?8?-4?3?-5?4?2?2?-2?6?9?2?3?-9=3738.0; Lösung: -3+1+8*-4*3*-5*4*2-2+-2*6*9-2+3--9=3738.0

### Länge = 15:

Rätsel: 6?-1?-9?6?1?-2?-9?-6?2?-7?7?-4?-6?-2?5?-4=8294.0; Lösung: 6--1--9*6-1+-2*-9*-6*2*-7+7*-4*-6*-2*5*-4=8294.0

Rätsel: -5?4?-6?-3?9?-8?8?-6?-9?3?9?-3?3?-6?-6?9=5487.0; Lösung: -5*4+-6*-3*9+-8:8--6*-9*3*9*-3+3*-6*-6*9=5487.0;

### Länge = 18:

Rätsel: 3?7?6?9?9?5?7?5?5?1?9?2?6?9?4?8?8?7=-115585.0

Lösung: 3-7*6*9*9*5*7+5:5+1+9+2*6*9*4*8+8+7=-115585.0

Rätsel: 3?7?6?9?9?5?7?5?5?1?9?2?6?9?4?8?8?7=-86938.0

Lösung: 3+7*6-9*9*5*7*5*5+1+9+2*6-9*4*8*8*7=-86938.0

### Länge = 19:

Rätsel: 3?2?3?1?2?7?6?4?3?4?4?8?7?9?5?4?5?7?2=13937.0

Lösung: 3+2*3*1*2*7+6*4+3*4*4*8*7*9*5*4:5*7+2=13937.0

Rätsel: *3?2?3?1?2?7?*6?4*?3?*4*?4?*8*?7?*9?5*?4?*5*?7?2=97181.0*

Lösung: *3*2*3+1+2+7*6+4**3*4**4**8**7*9+5**4*5**7:2=97181.0*

## 3.2 Beispiele gemäß meinen Erweiterungen:

### Länge = 5:

Rätsel: 9?2?5?-8?-5?7=37.0; Lösung: 9+2(5+-8--5)7=37.0

Rätsel: 3?1?7?-9?5?-9=192.0; Lösung: 3-1(7--9+5)-9=192.0

### Länge = 6:

Rätsel: 5?5?-8?-9?1?-4?-9=1008.0; Lösung: 5-5--8(-9-1+-4)-9=1008.0

Rätsel: 5?5?-8?-9?1?-4?-9=815.0; Lösung: 5+5(-8*-9:1*-4)-9=815.0

### Länge = 7:

Rätsel: 5?-2?3?2?-2?2?-5?4=180.0; Lösung: 5*-2(3+2)-2*2+-5*4=180.0

Rätsel: 5?-2?3?2?-2?2?-5?4=349.0; Lösung: 5--2(3+2*-2*2*-5)4=349.0

### Länge = 8:

Rätsel: 6?5?-1?9?-3?-9?6?8?5=8251.0; Lösung: 6^5--1(9+-3--9*6)8-5=8251.0

Rätsel: 6?5?-1?9?-3?-9?6?8?5=280176.0; Lösung: 6^5(-1+9:-3)-9+6*8*5=280176.0

## 4. Quellcode

### 4.1 Methode evalEquation:

```
public Double evalEquation(String equ){
        ArrayList<String> equArr = this.convertStringToArrayList(equ);
        if(equArr.contains("(") && equArr.contains(")")){
            int anzahlKlammernAuf = 0;
            int anzahlKlammernZu = 0;
            for(int g=0; g<equArr.size(); g++) {
                if(equArr.get(g).equals("(")) {
                    anzahlKlammernAuf++;
                    for(int i = g+1; i<equArr.size() ; i++){
                        if(equArr.get(i).equals("(")){
                            anzahlKlammernAuf++;
                        }else if(equArr.get(i).equals(")")){
                            anzahlKlammernZu++;
                            if(anzahlKlammernAuf == anzahlKlammernZu){
                                if(i+1<equArr.size()) {
                                    if(!listeRechenzeichen.contains(equArr.get(i+1))){
                                        equArr.add(i+1, "*");//soll vorherr passieren bevor die
klammern rausgekuerzt werden
                                    }
                                }
                                Double tempErgebniss = evalEquation(this.convertArrayListTo-
String(equArr, g+1, i-1));
                                if(tempErgebniss == null) {
                                    return null; //prueft ob ein valides Ergebniss gefunden wurde
                                }
                                equArr = this.shortEquation(equArr, String.valueOf(tem-
pErgebniss), g, i);
                                if(g-1>= 0) {
                                    if(listeRechenzeichen.contains(equArr.get(g-1)) == false){
                                        equArr.add(g,"*"); //nachgucken, ob es wirklich so funk-
tioniert
                                    }
                                }
```

```
                                System.out.println(this.convertArrayListTo-
String(equArr,0,equArr.size()-1));
                                i=equArr.size();
                            }
                        }
                    }
                    if(anzahlKlammernAuf != anzahlKlammernZu) {
                        return null;
                    }
                }
            }
        }
        if(equArr.contains("(") || equArr.contains(")")) {
            return null;
        }
        while(equArr.contains("^")) {
            for(int i=0; i<equArr.size(); i++){
                if(equArr.get(i).equals("^")){
                    double temp = Math.pow(Double.valueOf(equArr.get(i-1)),Double.val-
ueOf(equArr.get(i+1)));
                    equArr = this.shortEquation(equArr, String.valueOf(temp), i-1, i+1);
                }
            }
        }
        while(equArr.contains("*")  || equArr.contains(":")) {
            for(int i=0; i<equArr.size(); i++){
                if(equArr.get(i).equals("*")){
                    double temp = (Double.valueOf(equArr.get(i-1)) * Double.val-
ueOf(equArr.get(i+1)));
                    equArr = this.shortEquation(equArr, String.valueOf(temp), i-1, i+1);
                }else if(equArr.get(i).equals(":")){
                    double temp = (Double.valueOf(equArr.get(i-1)) / Double.val-
ueOf(equArr.get(i+1)));
                    if(temp%1 != 0) {
                        return null;
                    }
                    equArr = this.shortEquation(equArr, String.valueOf(temp), i-1, i+1);
                }
            }
        }
        while(equArr.contains("+")  || equArr.contains("-")) {
            for(int i=0; i<equArr.size(); i++){
                if(equArr.get(i).equals("+")){
                    double temp = (Double.valueOf(equArr.get(i-1)) + Double.val-
ueOf(equArr.get(i+1)));
                    equArr = this.shortEquation(equArr, String.valueOf(temp), i-1, i+1);
                }else if(equArr.get(i).equals("-")){
```

```
                    double temp = (Double.valueOf(equArr.get(i-1)) - Double.val-
ueOf(equArr.get(i+1)));
                    equArr = this.shortEquation(equArr, String.valueOf(temp), i-1, i+1);
                }
            }
        }
        if(equArr.size() == 1) {
            return Double.valueOf(equArr.get(0));
        }
        return null;
    }
```

## 4.2 Methode testRaetselOriginalMitKlammern;

```
public HashMap<Double, String> testRaetselOriginalMitKlammern(int currTiefe, int maxTiefe, String
oldEquation, String operator, int AnzahlOffeneKlammern) {//maxTiefe= Anzahl der Operatoren
        //Methode erstellen, welche eine zufällige Liste an Zahlen erstellt, welche für das ge-
samte Rätsel verwendet werden.
        if(currTiefe == maxTiefe) {
            String currEquation = oldEquation + operator + zahlen[currTiefe];
            Double ergebniss = this.evalEquation(currEquation);
            System.out.println(currEquation + "=" + ergebniss);
            if(ergebniss != null && !Gleichungen.containsKey(ergebniss) && !entfernteErgeb-
nisse.contains(ergebniss) ) {
                Gleichungen.put(ergebniss, currEquation);
            }else if(ergebniss != null) {
                Gleichungen.remove(ergebniss);
                if(!entfernteErgebnisse.contains(ergebniss)) {
                    entfernteErgebnisse.add(ergebniss);
                }
            }
            return Gleichungen;
            //Abbruchmethode schreiben
            //Ergebniss zur HashMap hinzufügen
        }else {
            String currEquation = oldEquation + operator + zahlen[currTiefe];
            Double ergebniss = 1.0;
            if(operator.equals(":")) {
                ergebniss = this.evalEquation(currEquation);
            }
            System.out.println(currEquation);
            if(ergebniss != null || AnzahlOffeneKlammern > 0) {
                currTiefe++;
                for(int i=0; i<rechenzeichenOriginal.length; i++) {
                    System.out.println("1");
                    Gleichungen.putAll(this.testRaetselOriginalMitKlammern(currTiefe, maxTiefe,
currEquation, rechenzeichenOriginal[i], AnzahlOffeneKlammern));
                }
                if(AnzahlOffeneKlammern<(maxTiefe-currTiefe-2)) {
```

```
                    int tempAnzahloffeneKlammern = AnzahlOffeneKlammern+1;
                    System.out.println("2");
                    Gleichungen.putAll(this.testRaetselOriginalMitKlammern(currTiefe, maxTiefe,
currEquation, "(", tempAnzahloffeneKlammern));
                }
                if(AnzahlOffeneKlammern>0 && !operator.equals("(")) {
                    int tempAnzahloffeneKlammern = AnzahlOffeneKlammern-1;
                    System.out.println("3");
                    Gleichungen.putAll(this.testRaetselOriginalMitKlammern(currTiefe, maxTiefe,
currEquation, ")", tempAnzahloffeneKlammern));
                }
            }
        }
        //Klammern mechanik einfügen;

        return Gleichungen;
    }
```

## 4.3 Methode testRaetselOriginal:

```
public HashMap<Double, String> testRaetselOriginalMitKlammern(int currTiefe, int maxTiefe, String
oldEquation, String operator, int AnzahlOffeneKlammern) {//maxTiefe= Anzahl der Operatoren
        //Methode erstellen, welche eine zufällige Liste an Zahlen erstellt, welche für das ge-
samte Rätsel verwendet werden.
        if(currTiefe == maxTiefe) {
            String currEquation = oldEquation + operator + zahlen[currTiefe];
            Double ergebniss = this.evalEquation(currEquation);
            System.out.println(currEquation + "=" + ergebniss);
            if(ergebniss != null && !Gleichungen.containsKey(ergebniss) && !entfernteErgeb-
nisse.contains(ergebniss) ) {
                Gleichungen.put(ergebniss, currEquation);
            }else if(ergebniss != null) {
                Gleichungen.remove(ergebniss);
                if(!entfernteErgebnisse.contains(ergebniss)) {
                    entfernteErgebnisse.add(ergebniss);
                }
            }
            return Gleichungen;
            //Abbruchmethode schreiben
            //Ergebniss zur HashMap hinzufügen
        }else {
            String currEquation = oldEquation + operator + zahlen[currTiefe];
            Double ergebniss = 1.0;
            if(operator.equals(":")) {
                ergebniss = this.evalEquation(currEquation);
            }
            System.out.println(currEquation);
            if(ergebniss != null || AnzahlOffeneKlammern > 0) {
                currTiefe++;
```

```
                for(int i=0; i<rechenzeichenOriginal.length; i++) {
                    System.out.println("1");
                    Gleichungen.putAll(this.testRaetselOriginalMitKlammern(currTiefe, maxTiefe,
currEquation, rechenzeichenOriginal[i], AnzahlOffeneKlammern));
                }
                if(AnzahlOffeneKlammern<(maxTiefe-currTiefe-2)) {
                    int tempAnzahloffeneKlammern = AnzahlOffeneKlammern+1;
                    System.out.println("2");
                    Gleichungen.putAll(this.testRaetselOriginalMitKlammern(currTiefe, maxTiefe,
currEquation, "(", tempAnzahloffeneKlammern));
                }
                if(AnzahlOffeneKlammern>0 && !operator.equals("(")) {
                    int tempAnzahloffeneKlammern = AnzahlOffeneKlammern-1;
                    System.out.println("3");
                    Gleichungen.putAll(this.testRaetselOriginalMitKlammern(currTiefe, maxTiefe,
currEquation, ")", tempAnzahloffeneKlammern));
                }
            }
        }
        //Klammern mechanik einfügen;

        return Gleichungen;
    }
```

## Klasse RaetselDatapack:

```
import java.util.ArrayList;
import java.util.HashMap;

public class RaetselDatapack {
    ArrayList<Double> ergebnisse;
    HashMap<Double, String> gleichungen;
    public RaetselDatapack(ArrayList<Double> pErgebnisse, HashMap<Double, String> pGleichungen) {
        ergebnisse = pErgebnisse;
        gleichungen = pGleichungen;
    }
    /**
     * This Methode is used to combine the Data of pData1 and pData 2
     * and return it afterwards as an Object containing both Datasets
     * the important thing to remember is that duplicates are beeing removed from the hashMap
     * but stay in the ArrayList for future identification
     * @param pData1
     * @param pData2
     * @return the combined Data of pData1 and pData2
     */
    public RaetselDatapack mergeDatapacks(RaetselDatapack pData1, RaetselDatapack pData2) {
        RaetselDatapack tempDatapack = new RaetselDatapack(new ArrayList<Double>(), new
HashMap<Double, String>());
        for(int i=0; i<pData1.ergebnisse.size(); i++) {
```

```
            while(pData2.ergebnisse.contains(pData1.ergebnisse.get(i))) {
                pData2.ergebnisse.remove(pData1.ergebnisse.get(i));
                pData2.gleichungen.remove(pData1.ergebnisse.get(i));
                pData1.gleichungen.remove(pData1.ergebnisse.get(i));
            }
        }
        tempDatapack.ergebnisse.addAll(pData1.ergebnisse);
        tempDatapack.ergebnisse.addAll(pData2.ergebnisse);
        tempDatapack.gleichungen.putAll(pData1.gleichungen);
        tempDatapack.gleichungen.putAll(pData2.gleichungen);
        return tempDatapack;
    }
}
```

## 4.4 Klasse MultithreadRaetselOriginal:

```
public class MultithreadRaetselOriginal implements Callable<RaetselDatapack>{
    int currTiefe;
    int maxTiefe;
    String oldEquation;
    String operator;
    int[] zahlen;

    String[] rechenzeichenOriginal = {"+", "-", "*", ":"};
    ArrayList<String> listeRechenzeichen = new ArrayList<String>();
    RaetselDatapack data = new RaetselDatapack(new ArrayList<Double>(), new HashMap<Double,
String>());
    public MultithreadRaetselOriginal(int pCurrTiefe, int pMaxTiefe, String pOldEquation, String
pOperator, int[] pZahlen) {
        currTiefe = pCurrTiefe;
        maxTiefe = pMaxTiefe;
        oldEquation = pOldEquation;
        operator = pOperator;
        zahlen = pZahlen;

        listeRechenzeichen.add("+");
        listeRechenzeichen.add("-");
        listeRechenzeichen.add("*");
        listeRechenzeichen.add(":");
        listeRechenzeichen.add("(");
        listeRechenzeichen.add(")");
        listeRechenzeichen.add("^");
    }

    public RaetselDatapack call() {//maxTiefe= Anzahl der Operatoren
        //Methode erstellen, welche eine zuf�llige Liste an Zahlen erstellt, welche f�r das ge-
samte R�tsel verwendet werden.
        if(currTiefe == maxTiefe) {
            String currEquation = oldEquation + operator + zahlen[currTiefe];
```

```
            Double ergebniss = this.evalEquation(currEquation);
        //  System.out.println(currEquation + "=" + ergebniss);
            if(ergebniss != null && !data.gleichungen.containsKey(ergebniss) &&
!data.ergebnisse.contains(ergebniss) ) {
                data.gleichungen.put(ergebniss, currEquation);
                data.ergebnisse.add(ergebniss);//wird trotzdem zur Liste hinzugef�gt, damit auch
die errechneten Ergebnisse aus den anderen Teilb�umen ber�cksichtigt werden k�nnen
            }else if(ergebniss != null) {
                data.gleichungen.remove(ergebniss);
                if(!data.ergebnisse.contains(ergebniss)) {
                    data.ergebnisse.add(ergebniss);
                }
            }
            return data;
            //Abbruchmethode schreiben
            //Ergebniss zur HashMap hinzuf�gen
        }else {
            String currEquation = oldEquation + operator + zahlen[currTiefe];
            Double ergebniss = 1.0;
            if(operator.equals(":")) {
                ergebniss = this.evalEquation(currEquation);
            }
//          System.out.println(currEquation);
            if(ergebniss != null) {
                currTiefe++;
                if(currTiefe == 1 && maxTiefe >= 8) {
                    ExecutorService es = Executors.newFixedThreadPool(6);
                    List<MultithreadRaetselOriginal> tasklist = new ArrayList<>();
                    for(int i=0; i<rechenzeichenOriginal.length; i++) {
                        MultithreadRaetselOriginal task = new MultithreadRaetselOriginal(1, zah-
len.length-1, "3", rechenzeichenOriginal[i], zahlen);
                        tasklist.add(task);
                    }
                    List<Future<RaetselDatapack>> resultList =null;
                    try {
                        resultList = es.invokeAll(tasklist);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    es.shutdown();
                    //System.out.println("Results are there");
                    RaetselDatapack data = new RaetselDatapack(new ArrayList<Double>(), new
HashMap<Double, String>());
                    for(int i = 0; i<resultList.size(); i++) {
                        Future<RaetselDatapack> future = resultList.get(i);
                        try {
                            RaetselDatapack result = future.get();
```

```
                        data = data.mergeDatapacks(data, result);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    } catch (ExecutionException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            }else {
                for(int i=0; i<rechenzeichenOriginal.length; i++) {
                    RaetselDatapack tempDatapack2 = this.testRaetselOriginal(currTiefe, max-
Tiefe, currEquation, rechenzeichenOriginal[i]);
                    data = data.mergeDatapacks(data, tempDatapack2);
                }
            }
        }
    }
    return data;
}
public RaetselDatapack testRaetselOriginal(int currTiefe, int maxTiefe, String oldEquation,
String operator) {//maxTiefe= Anzahl der Operatoren
    RaetselDatapack tempDatapack = new RaetselDatapack(new ArrayList<Double>(), new
HashMap<Double, String>());
    //Methode erstellen, welche eine zuf□llige Liste an Zahlen erstellt, welche f□r das ge-
samte R□tsel verwendet werden.
    if(currTiefe == maxTiefe) {
        String currEquation = oldEquation + operator + zahlen[currTiefe];
        Double ergebniss = this.evalEquation(currEquation);
        //System.out.println(currEquation + "=" + ergebniss);
        if(ergebniss != null && !tempDatapack.gleichungen.containsKey(ergebniss) && !temp-
Datapack.ergebnisse.contains(ergebniss) ) {
            tempDatapack.gleichungen.put(ergebniss, currEquation);
            tempDatapack.ergebnisse.add(ergebniss);//wird trotzdem zur Liste hinzugef□gt,
damit auch die errechneten Ergebnisse aus den anderen Teilb□umen ber□cksichtigt werden k□nnen
        }else if(ergebniss != null) {
            tempDatapack.gleichungen.remove(ergebniss);
            if(!tempDatapack.ergebnisse.contains(ergebniss)) {
                tempDatapack.ergebnisse.add(ergebniss);
            }
        }
        return tempDatapack;
        //Abbruchmethode schreiben
        //Ergebniss zur HashMap hinzuf□gen
    }else {
        String currEquation = oldEquation + operator + zahlen[currTiefe];
        Double ergebniss = 1.0;
        if(operator.equals(":")) {
```

```
                ergebniss = this.evalEquation(currEquation);
            }
    //      System.out.println(currEquation);
            if(ergebniss != null) {
                currTiefe++;
                for(int i=0; i<rechenzeichenOriginal.length; i++) {
                    RaetselDatapack tempDatapack2 = this.testRaetselOriginal(currTiefe, maxTiefe,
currEquation, rechenzeichenOriginal[i]);
                    tempDatapack = tempDatapack.mergeDatapacks(tempDatapack, tempDatapack2);
                }
            }
        }
        //Klammern mechanik einf�gen;

        return tempDatapack;
    }
```