

# Inhaltsverzeichnis

<b>1</b>	<b>Programm</b>	<b>2</b>
1.1	Einleitung . . . . .	2
1.2	Idee . . . . .	3
1.3	Implementation in Java . . . . .	3
1.3.1	Berechnen der Sektionen . . . . .	3
1.3.2	Berechnen der durchschnittlichen Farbe . . . . .	4
1.3.3	Erstellen einer Datenbank . . . . .	4

# 1 Programm

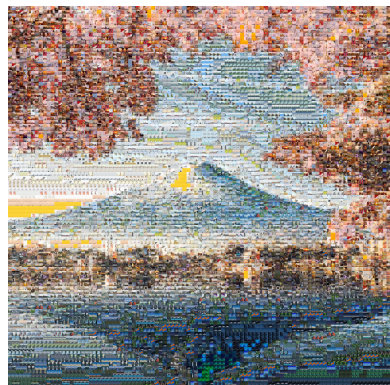
## 1.1 Einleitung

Um die Methoden der Threads in einer Realen Situation zu nutzen, habe ich mich entschlossen ein Programm, welches stark von Threads profitieren kann, zu programmieren.

Ich habe mich für ein Programm entschieden, mit dem man ein Bild mit vielen weiteren Bildern rekrieren kann. Es wird demnach ein Mosaik aus Bildern erstellt.



(a) Input



(b) Output

Abbildung 1: Funktionsweise des Programmes

Die anwendung von Threads kommt in dem Programm in vielen Stellen vor. Im folgen werde ich mich auf den Algorythmus der Bild analyse und verarbeitung beziehen. Andere aspekte, wie die implementation des Testmodus und andere Features, die in der App vorhanden sind, werden kurz im Anhang erwähnt.

## 1.2 Idee

In der genaueren Betrachtung, muss das Programm das eingegebene Bild vereinfachen. Dies wird durch eine Unterteilung des Bildes in Sektoren erreicht. Die Sektoren sind auch die zukünftigen Stellen für die ausgewählten Bilder. Es wird die durchschnittliche Farbe der Sektoren berechnet. Das selbe vorgehen wird auch auf die ausgewählten Bilder angewendet. Dabei ist ein Bild ein Sektor. Dieses vorgehen erlaubt es auch, Datenbanken (im JSON-Format) zu erstellen, da sich die durchschnittliche Farbe der ausgewählten Bilder nicht ändern wird. Anschließend wird mit einem Algorithmus die passenden Bilder für die einzelnen Sektoren berechnet. Dabei unterliegt der Algorithmus der Beschränkung, dass der Nutzer wählen kann, wie häufig ein Bild vorkommen darf. Nach dem berechnen der benötigten Bilder müssen diese für die Verwendung angepasst werden. Es soll schließlich nicht ein 10x10px großer Teil aus einem 4000x3000px Bild verwendet werden. Dazu wird das Bild erst skaliert und schließlich zurechtgeschnitten. Dabei liegt der Fokus darauf, immer die Mitte des Bildes zu treffen. Das Programm arbeitet die Schritte nachfolgend ab und in dieser Reihenfolge werde ich das Programm nachfolgend erklären.

## 1.3 Implementation in Java

### 1.3.1 Berechnen der Sektionen

Das berechnen der Sektionen ist eigentlich ziemlich geradeaus. Die Größe des Bildes ist bekannt und in wie viele Teile dieses aufgeteilt werden soll. Dazu wird zb. die Breite des Bildes mit der vertikalen Anzahl an Sektoren geteilt. Das Problem liegt darin, dass die Breite oder Höhe des Bildes nicht ein redundant von den gewünschten Sektoren ist. Um das Problem anzugehen wird ersterer Schritt nur mit Integers ausgeführt und der Rest berechnet wird. Übergeblieben sind wie viele Spalte auf der X- und Y-Achse gemacht müssen und der Rest der über bleibt. Im nächsten Schritt werden die Größen der Sektoren in Pixeln berechnet. Dies erfolgt dadurch, dass zwei Arrays für die X- und Y-Achse mit der Anzahl an benötigten Spalten gefüllt werden. Ein weiteres Array mit der Anzahl der Spalten wird aufsteigend nummeriert. Das Array wird von einer Methode gemischt. Eine abschließende Schleife wird mit der Anzahl des Restes wiederholt und bei jedem Schritt wird ein dann zufälliger Wert aus den Array als Index für das Array mit der Größe der Sektoren genutzt. Das nun zufällige Segment wird um 1 erhöht. Die berechneten Werte werden in einem "splitObj" gespeichert. In diesem werden die Werte in unterschiedlichsten Arten wie z.B. Koordinaten gespeichert.

Diese Koordinaten werden im nächsten Schritt benutzt, um ein zweidimensio-

nales Array an “BufferedImages” aus dem Original Bild zu extrahieren. Dazu wird die “getSubimage()” Methode der Klasse “BufferedImage” verwendet.

### 1.3.2 Berechnen der durchschnittlichen Farbe

Das berechnen der durchschnittlichen Farbe von Segmenten und Bildern ist die erste Funktion, welche Threads nutzt. Implementationen von mir, welche Threads benutzen, bestehen meistens aus zwei Klassen. Eine Klasse, welche die Threads erstellt und verwaltet und eine weitere, welche den Code zum berechnen der jeweiligen Anforderung hat. In diesem Falle das Berechnen der durchschnittlichen Farbe. Die Klasse “computeAverageColor” besteht aus zwei Methoden. Eine zum Berechnen der Segmente und eine weitere für die gewählten Bilder. Der einzige Unterschied der beiden Methoden ist eine Sicherheitsfunktion in der Methode der Bilder welche wartet, bis genügend RAM verfügbar ist, bevor es das Bild in den RAM lädt. Dies ist wichtig, da mehrere Threads gleichzeitig die Bilder im RAM zum Berechnen halten müssen. Je nach RAM Configuration und Bildergröße kann dies zu Komplikationen führen. Jeder der Threads ruft die Methode “getAverage()” der Klasse “calculateAverage” auf. Die Methode verlangt ein “BufferedImage” und ein enum “calculateAverage.Method”, welches unterschiedliche Genauigkeitsstufen zum berechnen beinhaltet. Innerhalb der Methode “getAverage()” wird erst mit Hilfe des enums ein int definiert, welches beschreibt wie die nachfolgenden for-Schleifen erhöht werden. Weiterhin gibt es drei “long”. Diese speichern den gesamten rot, blau und grün wert. Anschließend wird das Bild in der vorherigen definierten Schrittgröße durchgegangen und die rot, grün und blau werte des Pixels gespeichert. Zum Schluss wird ein neues “Color” Object mit den gesammelten werten geteilt durch die Anzahl der Pixel.

### 1.3.3 Erstellen einer Datenbank

Dieser Schritt wird immer dann durchgeführt, wenn im Programm Bilder ausgewählt wurden, oder eine speicherbare Datenbank erstellt wird. Wenn Datenbanken ausgewählt wurden, wird die eventuell erstellte Datenbank diesen hinzugefügt. Die Klasse “DatabaseObj” benötigt zwei Argumente. Eine Liste an Speicherorten der jeweiligen Bilder und ein Array an berechneten durchschnittlichen Farben. Innerhalb des “DatabaseObj” werden die beiden Werte der Listen in “fileAndColor” Objekten gespeichert. Wie der Name impliziert beinhalten die Objekte den Pfad des Bildes und den dazugehörigen Farbwert. Durch dieses Vorgehen kann ein BinarySort Algorithmus auf das neue Array angewendet werden. dazu hat die “fileAndColor” Klasse das “Comparable” interface implementiert. Für spätere Verwendung gibt es in den Objekten auch

noch einen Zähler, welcher speichert, wie häufig das Bild in der Berechnung verwendet wurde. Alle Variablen der Klasse “fileAndColor” sind durch getter-setter-Methoden aufzurufen. Diese haben jedoch die Besonderheit, dass sie das “synchronized” Schlüsselwort besitzen ([Hyperlink zu Kapitel Threads](#)).

Besonders werde ich auf die Methode “compareTo()”, des Interfaces Comparable eingehen. In der Methode werden zwei Farben verglichen. Welche größer und kleiner ist. Die Schwierigkeit liegt darin ein Ergebnis zu bestimmen, wenn vier Werte verglichen werden müssen. Der alpha-Wert, rot-Wert, grün-Wert und blau-Wert. Um einen einfachen und zuverlässigen Vergleich zu machen, nutze ich die Methode “getRGB()” der “Color” Klasse. Interessant ist jedoch, was diese Methode macht. In der Klasse “Color” werden alle vier Werte in einem einzigen Integer gespeichert. Das liegt an der Genauigkeit, womit Farben gemischt werden. Jeder Wert befindet sich in einem Bereich von 0 bis 255. Dies entspricht genau einem Byte. Ein Integer wiederum besteht aus vier Bytes. Die vier Farbwerte werden sozusagen in die vier Bytes des Integers eingesetzt. In der Reihenfolge “ARGB”. (Bild von Casting Tabelle) Durch dieses Vorgehen wird aus vier Werten ein zuverlässiger Wert erstellt, welcher zum Sortieren geeignet ist.