Conversie imagini grayscale in imagini binare

# Introducere

Cu toate ca exista multe biblioteci care implementeaza deja lucrul cu imaginile (e.g. AWT – built-in, ImageJ, OpenCV), in acest proiect voi folosi doar cateva obiecte din pachetul java.awt, implementarile fiind facute cu secvente de cod ce lucreaza la nivel de pixel.

Pentru a respecta in totalitate tema, am ales sa implementez o aplicatie in linia de comanda de la zero si nu am folosit vreun framework existent. Totusi, cu mici ajustari aplicatia creata poate fi intrdousa intr-una Spring Boot si astfel folosind cateva module web si adaugand o interfata bazata pe JSP-uri, ea poate fi deployata pe un server online.

# Descrierea aplicatiei cerute

Pornind de la cerintele de implementare, mai exact de la cerinta “3. Include in totalitate conceptele POO – incapsulare, mostenire, polimorphism, abstractizare”, am dedus ca aplicatia trebuie sa fie „cat mai” abstracta, astfel incat sa poata sa fie extinsa pentru oricare din temele propuse catre implementare.

Astfel, pentru tema asignata mie „Convert Gray-Scale Image to Binary image (Static Threshold)”, tema care ar putea fi implementata intr-o singura clasa urmarind un model secvential de programare, dar care nu ar mai putea fi extinsa, am ales sa creez o ierarhie de clase definita mai departe la punctul 4).

Aceasta conversie presupune existenta la intrare a unei imagini in tonuri de gri, deci aici pe langa validarile standard din cerintele de implementare, se mai adauga o verificare ca imaginea sa fie grayscale.

# Partea teoretica

Fisierul sursa este o image de 24 de biti RGB cu extensia BMP, extensie specifica sistemului de operare Microsoft Windows, care caraterizeaza bitmap-urile. Aceste bitmap-uri reprezinta matrice de pixeli, caracterizate de numarul de pixeli (inaltime – latime) si de densitatea informatiei per pixel.

Fisierul de iesire este o imagine binara, deci alb-negru (monocroma).

Cea mai buna implementare ar trebui sa duca imaginea de intrare de 24 biti per pixel, intr-o imagine cu 1 bit per pixel (unde 0 – negru, 1 – alb).

Bitmap-ul de 24 biti RGB contine astfel cate 8 biti pentru fiecare componenta (rosu, verde, albastru). Conditia ca imaginea sa fie gri este ca cele 3 componente sa fie egale, de aici rezultand si metoda de implementare a uneia dintre validari (si totodata realizarea ca imaginea RGB 24 biti de intrare ar putea fi de 8 biti, dar nu se mai respecta prima cerinta de implementare).

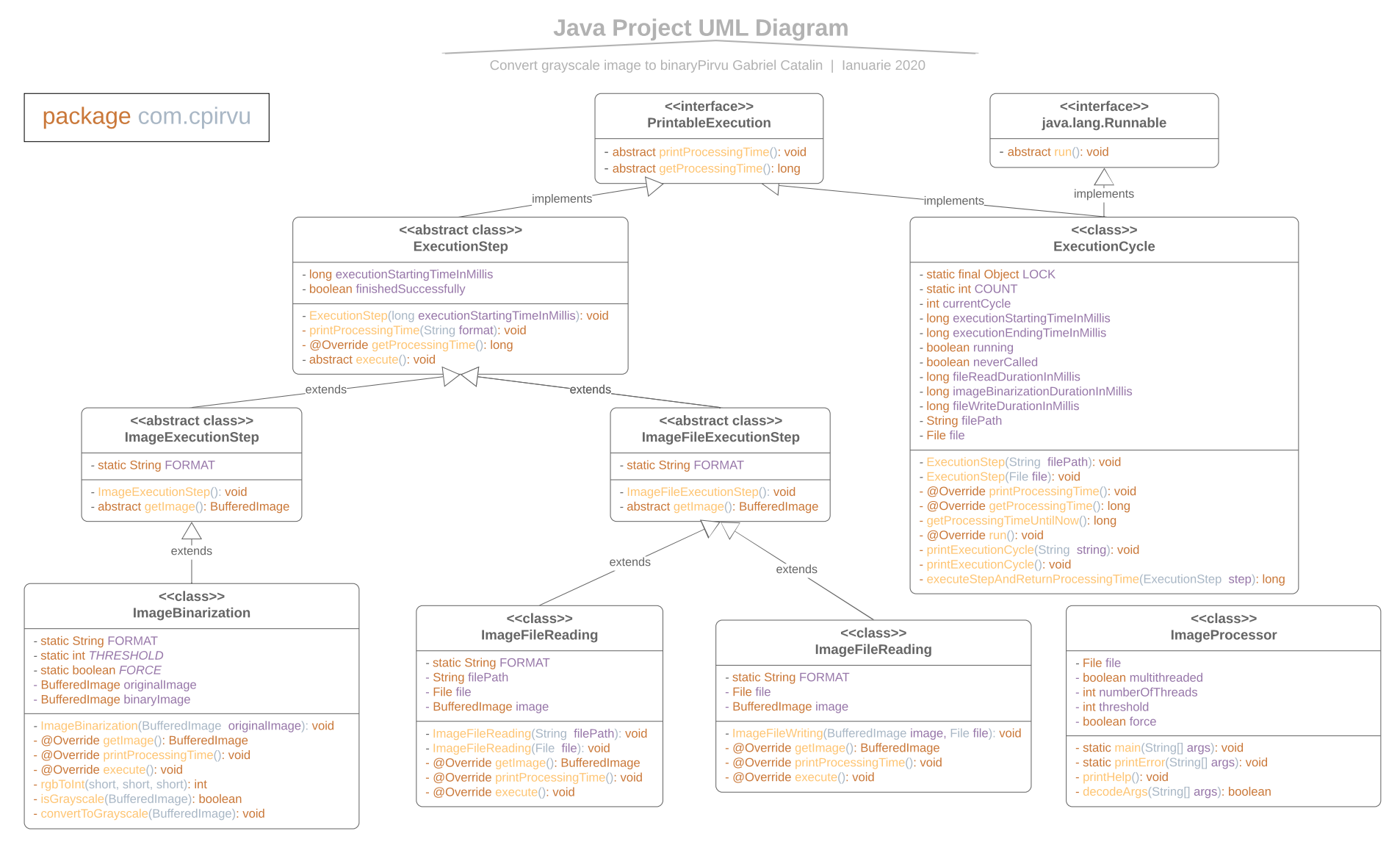
De asemenea, metoda „static threshold” presupune existenta unui prag care, daca este depasit de intensitatea pixelului curent, atunci el va deveni negru in imaginea finala, altfel va deveni alb. Aceasta metoda nu ia in considerare si relatia dintre pixeli, de aici si obtinerea unei imagini care poate pierde mult din continutul celei initiale

# Descrierea implementarii

Dupa cum am mentionat, motivele alegerii unei aplicatii CLI este acela de a respecta in totalitate cerintele. Pentru aceasta, am ales sa utilizez java 8. Din pachetul standard, am utilizat java.awt.image.BufferedImage (pentru stocarea imaginii cu toate elementele ei) si javax.imageio.ImageIO (pentru operatiile de citire a unei imagini dintr-un fisier, dar si scriere a unei imagini intr-un fisier.

Aproape tot codul este documentat cu javadoc, de aceea recomand parcurgerea acestuia pentru vizualizarea unei imagini mai detaliate asupra proiectului.

Diagrama UML a aplicatiei este urmatoarea:



In dezvoltarea aplicatiei, am pornit prin crearea interfetei PrintableExecution care defineste acele clase ale caror executii pot fi aflate sau scrie in consola.

Mai departe, tinand cont de faptul ca la intrare putem avea mai mult de un fisier, am creat un ciclu de executie marcat de ExecutionCycle, care implementeaza interfata Runnable si e creat astfel incat un ciclu de executie sa se ocupe de transformarea in binar a unei imagini. De asemenea, implementeaza si PrintableExecution, pentru ca ne intereseaza durata acestuia.

Un ExecutionCycle este format din mai multi pasi (ExecutionStep’s) care pot fi categorizati in doua tipuri: cei care se ocupa cu transformarea imaginii (ImageExecutionStep) si cei care se ocupa cu fiserele imagine - citire/scriere (ImageFileExecutionStep).

Clasele concrete care definesc acesti pasi sunt: ImageBinarization, ImageFileReading si ImageFileWriting.

Astfel cunoscuta arhitectura high-level, putem vorbi de modulele individuale, dupa cum se executa apelurile interne, pornind din main():

## ImageProcessor

Intrarea, ca in orice aplicatie, se face prin metoda public static void main(String[] args). Aici am optat pentru crearea unui obiect de tipul ImageProcessor, pentru a putea lucra mai simplu cu obiecte dupa ce decodific argumentele de intrare.

Apelul metodei private boolean decodeArgs(String[] args) are ca rezultat un boolean care valideaza argumentele de intrare. Astfel, aplicatia permite existenta mai multor flag-uri venite prin intermediul liniei de comanda pentru personalizarea executiei, precum:

(parantezele drepte „[]” marcheaza optionalitatea flagului)

* -P <path> => calea catre fisierul BMP 24bit RGB sau un director care contin astfel de fisiere (metoda verifica doar existenta path-ului in sistemul de fisiere)
* [-M <numberOfThreads>] => aplicatia poate folosi pana la „numberOfThreads” fire de executie in procesarea imaginilor (daca doar „-M” este prezent, atunci se vor folosi 5 threaduri)
* [-T <staticThreshold>] => pragul uitilizat in procesul de binarizare (daca doar „-T” este prezent, atunci se va folosi pragul de 127)
* [-F] => „force”, flag ce marcheaza deblocarea lucrului cu fisiere colore, sub acelasi format 24 bit RGB, conversia intermediara la grayscale facandu-se cu metoda luminozitatii)

Ordinea parametrilor este irelevanta.

Prezenta duplicatelor, a argumentelor necunoscute, invalide, cele care nu se incadreaza in limitele impuse sau neintroducerea argumentului „-P” urmate de o cale duce la apelul metodei private static void printError(String[] args), care afiseaza parametrii de intrare si recomanda utilizarea apelului folosind „help” ca argument de intrare. Apelul private void printHelp() duce la scrierea in linia de comanda a mai multor informatii privind executia aplicatiei.

Dupa validarea argumentelor de intrare, metoda main() seteaza variabilele statice THRESHOLD si FORCE din ImageBinarization.

Daca la calea de intrare este un fisier, ciclul de executie va fi sincron, pe acelasi main thread.

Altfel, daca este un director si flagul „-M” a fost setat, atunci pentru fiecare fisier (in afara de cele asunse filtrate cu o expresie lambda) va fi apelat un alt ciclu de executie, planificat de un pool de threaduri, ScheduledThreadPoolExecutor cu numarul de thread definit de parametrul numberOfThreads - default sau definit de utilizator).

Daca se omite „-M”, atunci fisierele din director vor fi supuse procesorului de binarizare in mod sincron, pe main thread.

## ExecutionCycle

Aici, in metoda public void run(), se face trecerea de la o imagine grayscale la una binara, cu inregistrarea startului si finalului executiei, dar si durata fiecarui pas individual.

O executie se face prin apeluri inaltuite a metodei public void execute():

* ImageFileReading#execute()
* ImageBinarization#execute()
* ImageFileWriting#execute()

Daca una dintre executii nu se termina cu succes, atunci intreg ciclu se opreste fiind afisate mesaje de eroare.

Aceasta insiruire de executii se face prin apelul metodei private long executeStepAndReturnProcessingTime(), a carui argument este un ExecutionStep, metoda ce afiseaza totodata si timpul de executie individual.

Clasa mai contine si un lock, pentru acele executii ce folosesc procesarea paralela. Acest LOCK este un obiect static, sharuit de toate obiectele de tip ExecutionCycle si pe care realizam sincronizarea. Blocurile synchronized din cod sunt folosite doar pentru ca afisarea in linia de comanda sa fie una cat mai usor de citit, pentru ca exista posibilitatea de a fi apelate de pe fire de executie diferite metodele System.out.println().

## ImageFileReading

Aceasta implementare a metodei execute() relizeaza deschiderea fisierului si conversia acestuia intr-o BufferedImage, cu validarea fisierului de intrare:

* extensia sa fie „bmp”
* channel-ul utilizat de bitmap sa fie pe 24 de biti

## ImageBinarization

Metoda execute() testeaza mai intai daca imaginea este in tonuri de gri. In cazul in care nu este, iar flagul FORCE este setat, aceasta se converteste la gri folosind metoda luminozitatii.

Testarea grayscale se realizeaza prin parcurgerea matricei de pixeli si verificarea ca componentele red, green, blue ale fiecarui pixel sa fie egale.

Conversia la grayscale se face dupa formula 0.2126 \* red + 0.7152 \* green + 0.0722 \* blue, una dintre reprezentarile grayscale percepute foarte bine de ochiul uman, verdele avand cea mai mare pondere.

In cazul in care imaginea nu este grayscale si flag-ul nu este setat, atunci se iese din executie.

Conversia la binar sa face prin parcurgerea matricei si asginarea 0 (black) pixelului in cazul in care depaseste THRESHOLD, altfel asgineaza 255 (white). De asemenea, folosim si metoda ajutatoare rgbToInt() care accepta 3 parametrii de tipul short corespunzatorii elementelor r/g/b si returneaza un int ce contine reprezentarea rgb in timpul intreg (biti 24-16 => red, biti 15-8 => green, bitii 7-0 => blue).

## ImageBinarization

Metoda execute realizeaza scrierea imaginii procesate la pasul anterior, la nou fisier de intrare. Fisierul este scris in aceasi locatie, numele acestuia continand acum la final „\_BINARIZED”

# Evaluare performante

Folosind mai multe imagini, printre care si imagini care nu sunt valide (dpdv extensie, format, etc.), in modul sincron in medie se obtin ~60ms pe ciclu de executie. Conversia la catre grayscale, daca are loc, dureaza aproximativ ~15ms, deci putem deduce ca aproximativ 45ms sunt suficiente pentru binarizarea unei imagini RGB 24bit. De altfel, dimensiunea initiala a imaginii, care variaza cel mai mult in general, este si factorul care influenteaza cel mai mult timpul de executie.

Pe pasi individuali, timpul de executie este urmatorul:

* Reading: 9.5ms
* Binarization: 39.5ms
* Writing: 11ms

De asemenea, putem observa ca primul pas al primului ciclu, cel de Reading, dureaza mai mult decat deobicei (100~300 ms), asta datorita lazy-loading-ului prezent in pachetul javax.imageio.

Pentru modul de executie asincrona, durata unui ciclu de executie creste pana la aproximativ 80~100 ms (cu alegerea unui numar de threaduri optim – metoda try and fail), asta datorita faptului ca startul unui fir de executie este dependent de executia celorlalte fire, dar si ca apelul System.out.println() (= sincrona) este cea pe care o vrem sincrona in anumite zone in care se apeleaza secvential de mai multe ori (explicata mai sus in modulul ExecutionCycle).

Pentru un numar mai mare de fisiere de intrare, modul multi-threading este mult mai rapid.

# Instalare, rulare locala si documentatia

Intregul proiect, inclusiv fisierele .class generate, artifactul .jar, documentatia javadoc, dar si acest document se gasesc pe contul meu de github (<https://github.com/PirvuCatalin/image-processor>).

Acolo recomand urmarirea documentatiei javadoc generate pentru o mai buna intelegere a proiectului, dar si citirea readme-ului care va cuprinde cele ce urmeaza.

O sa atasez totusi o arhiva cu continutul proiectului, acestui document, in orice eventualitate.

Pasi pentru a testa folosind jar-ul si fiserele de testare:

1. Deschideti un terminal (MacOS/Linux) sau CMD (Win)
2. Navigati la /out/artifacts/ImageProcessor\_jar (cd ...)
3. Rulati comanda: java -jar ImageProcessor.jar -P "../../test"

Pasi pentru a testa folosind fisierele .class generate si fiserele de testare:

1. Deschideti un terminal (MacOS/Linux) sau CMD (Win)
2. Navigati la /out/production/ImageProcessor (cd ...)
3. Rulati comanda: java com.cpirvu.ImageProcessor -P "../../test"

Pentru mai multe informatii de utilizare, rulati „java -jar ImageProcessor.jar help”.

Un exemplu de rulare care utilizeaza un singur fisier: java -jar ImageProcessor.jar -P „../../test/lena\_gray\_24bit.bmp”.

Un exemplu de rulare care utilizeaza toti parametrii disponibili: „java -jar ImageProcessor.jar -P „../../test” -F -M 4 -T 150”.

# Bibliografie

[1] OCA Java SE 8 Programmer I Study Guide – Jeanne Boyarsky and Scott Selikoff (<https://www.youracclaim.com/badges/12be71d0-b2f8-4ad8-88ea-45107c3ffe33>)

[2] OCP Java SE 8 Programmer II Study Guide – Jeanne Boyarsky and Scott Selikoff

[3] Bitmaps Guide <http://paulbourke.net/dataformats/bitmaps/> - Paul Bourke

[4] <https://en.wikipedia.org/wiki/Grayscale>

[5] <https://en.wikipedia.org/wiki/Thresholding_(image_processing)>