Tema 2 - Selfie

Obiective

- fundamentarea practică a constructorilor și a agregării/moștenirii
- dezvoltarea unor abilități de bază de organizare și design orientat-obiect
- respectarea unui stil de codare si de comentare
- utilizarea unei arhitecturi bazate pe mesaje

Scenariu

Silvia, pasionata de selfie-uri, doreste sa vada care este procesul dintr-un aparat de fotografiat. Ea apeleaza la niste prieteni (Steve, Bill si Linus) studenti din anul 2 de la Facultatea de Calculatoare pentru a-i explica modul de functionare. Acestia propun o simulara minimalista a principalelor componente ce compun procesul de fotografiere.

Cei trei organizeaza simularea in trei etape: precaptura, captura si postcaptura.

Bill spune ca atunci cand vrei sa faci o fotografie poti sa utilizezi optiunea de **zoom** (dintr-un peisaj vrei sa capturezi doar o regiune) si optiunea de **blit** (care poate fi ON, OFF, AUTO) si influenteaza luminozitatea peisajului.

Pasionat de optica, Bill mai spune ca un aparat cu obiectiv captureaza initial imaginea rasturnata, iar apoi o salveaza in pozitie normala.

Steve, fiind in trend cu tehnologia, propune trei filtre: blur, sepia si alb/negru in partea de postcaptura.

Tragand linie, Linus observa un numar de componente: zoom, blit, captura imaginii rasturnate, inversarea imaginii, filtrul blur, filtrul sepia si filtrul alb/negru. El propune o arhitectura a aplicatiei bazate pe schimb de mesaje intre aceste componente. De asemenea, avand in vedere ca simularea se va face pe un calculator, el adauga doua componente in plus: incarcarea peisajului in memorie si salvarea pozei pe disk.

Dupa rularea simularii cu diversi parametrii, Silvia poate observa etapele intermediare dintr-un aparat de fotografiat.

Arhitectura aplicatiei

Etapele descrise mai sus: ImageLoader, ImageSaver, Zoom, Flash, RawPhoto, NormalPhoto, Sepia, BlackWhite, Blur vor fi implementari ale unei clase abstracte Component:

```
abstract class Component {

   TaskType type;

   abstract Message notify(Message message);

}
```

unde TaskType este un enum cu tipurile de task-uri:

```
enum TaskType {

   LOAD, SAVE, ZOOM, FLASH, RAW, NORMAL, SEPIA, BLACK_WHITE, BLUR

}
```

Mesajele care se trimit au la baza clasa abstracta Message.

```
abstract class Message {

   TaskType type;

}
```

Fiecare componenta va primi un tip de mesaj si va intoarce un alt tip de mesaj cu raspunsul corespunzator.

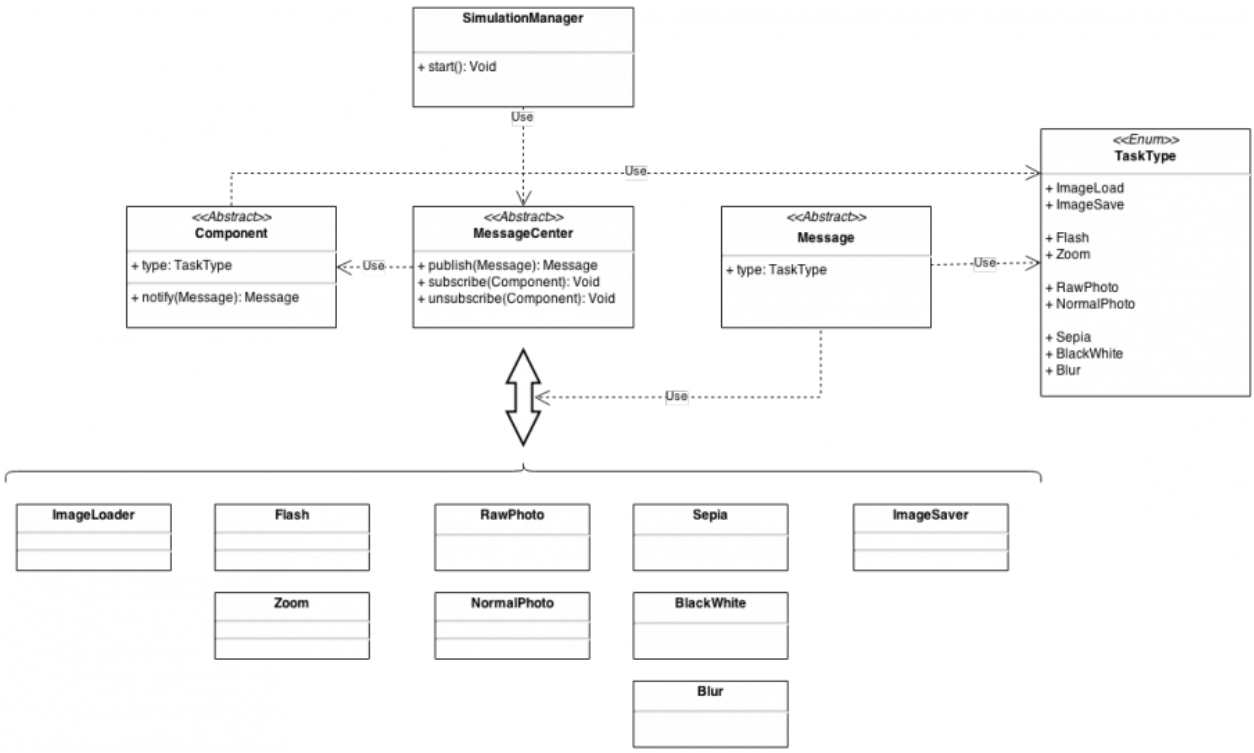
Exemplu:

ImageLoader - primeste un mesaj in care se afla numele fisierului pe care trebuie sa-l incarce si intoarce dimensiunea imaginii si matricea de pixeli.
Blur - primeste dimensiunea imaginii si matricea de pixeli si intoarce aceeasi dimensiunea a imaginii si matricea de pixeli modificata conform algoritmului de blur.

Componentele din punctul lor de vedere nu stiu decat sa rezolve un anumit tip de task si se vor inregistra la un MessageCenter pentru a primi task-uri incapsulate in clasa Message. Va exista o ierarhie de mesaje in functie de tipul de task.
Simulation manager va utiliza metoda publish de la MessageCenter pentru a rezolva un task.

MessageCenter-ul il va trimite catre o componenta care stie sa rezolve acel tip de task utilizand metoda notify.

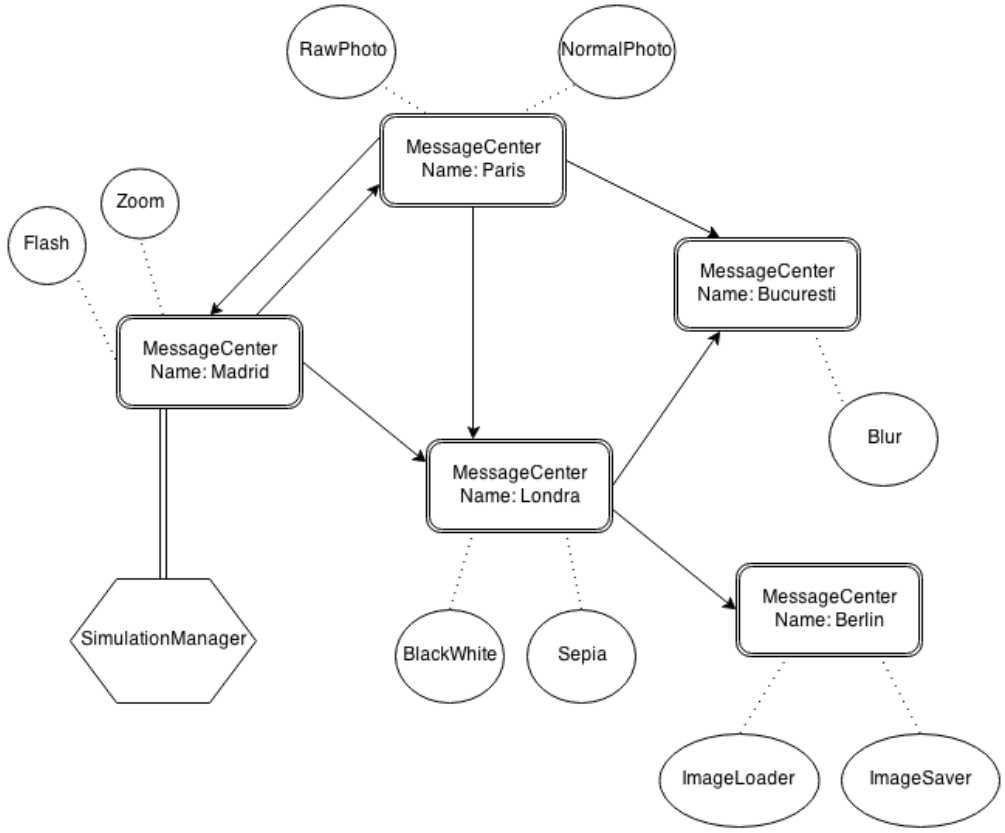
O imagine de ansamblu al unui MessageCenter:



In exemplul de mai sus, avem cazul in care un MessageCenter contine toate tipurile de componente. Insa, noi vrem sa avem MessageCenter-uri specializate pe anumite componente. Asadar, vom construi o retea de MessageCenter-uri si practic, fiecare MessageCenter va avea o lista de vecini la care poate apela daca el nu are componenta corespunzatoare pentru task-ul respectiv.

Ca imagine de ansamblu, vom avea un SimulationManager care initial va construi reteaua de MessageCentere, le va asocia componentele corespunzatoare si va crea legaturile dintre ele. El va interoga mereu un singur MessageCenter (primul declarat in fisierul de configurare).
Apoi, va citi comenzi de la standard input si va trimite task-uri MessageCenter-ului mentionat anterior, iar acesta va incerca sa gaseasca o componenta care stie sa rezolve task-ul respectiv. Initial, se va uita la componentele sale si daca contine o componenta compatibila atunci ii va trimite mesajul acestei componente. Altfel, parcurge lista de vecini si trimite mesajul acestora pana gaseste un raspuns.

Exemplu de retea:



Fisierul de intrare pentru reteaua de mai sus contine pe prima linie numarul de centre de mesagerie.
Apoi pe urmatoare n linii, avem pe prima pozitie numele centrului, urmat de componentele sale.
Pe urmatoarele(ultimele) n linii, avem pe prima pozitie tot numele centrului, urmat de vecinii sai in reteaua de centre.

```
5
Madrid Zoom Flash
Paris RawPhoto NormalPhoto
Bucuresti Blur
Londra BlackWhite Sepia
Berlin ImageLoader ImageSaver
Madrid Paris Londra
Paris Madrid Londra Bucuresti
Bucuresti
Londra Bucuresti Berlin
Berlin
```

Dupa cum putem observa, Bucuresti si Berlin nu au niciun vecin. Paris poate comunica cu Madrid pentru ca Madrid este in lista de vecini a lui Paris si la fel si Madrid poate comunica cu Paris

deoarece Paris este in lista de vecini a lui Madrid. Londra poate trimite mesaje catre Bucuresti si Berlin, dar invers nu se poate.

SimulationManager va comunica mereu cu primul MessageCenter din lista de centre. In cazul de mai sus cu Madrid.

Descrierea componentelor

Urmeaza o descriere detaliata a algoritmilor din fiecare componenta.

Lista cu tipurile de mesaje:
- MessageLoad
- MessageSave
- MessageImage
- MessageZoom
- MessageFlash
- MessageSuccess

ImageLoader si ImageSaver

ImageLoader si **ImageSaver** sunt in scheletul de cod deja implementate pentru a va usura munca si pentru a exista o uniformizare a modului in care se salveaza imaginile.
ImageLoader va primi un mesaj de tipul **MessageLoad** ce va contine un string cu calea catre imagine si returneaza un mesaj de tipul **MessageImage** ce contine o matrice de pixeli tridimensionala si dimensiunea imaginii:

```
int height;

int width;

int[][][] pixels;
```

Pentru o imagine cu dimensiunile 1024 px latime si 768 px inaltime, pixels va fi new int[768][1024][3] pentru ca avem 768 de randuri, 1024 de coloane si fiecare pixel are trei componente red, green si blue (RGB) cu valori cuprinse intre 0 si 255. Este necesar ca orice operatie care depaseste 255 sau este sub 0 sa fie adusa la valoarea maxima si respectiv minima pentru o obtine rezultatul corect. Am ales tipul int ca si reprezentare a matricei pentru a nu avea probleme de overflow.

pixels[0][0][0] va fi cantitatea de rosu din coltul stanga sus al pozei.
pixels[0][1023][2] va fi cantitatea de albastru al ultimului pixel de pe primul rand.

Pe de alta parte, componenta **ImageSaver** va primi un mesaj de tipul **MessageSave** ce va contine atat calea imaginii, unde va trebui sa salveze imaginea, cat si matricea de pixeli si dimensiunile acesteia. Va returna un mesaj de tipul MessageSuccess prin care va confirma faptul ca a putut salva imaginea.

Zoom

Componenta **Zoom** va primi un mesaj de tipul **MessageZoom** ce va contine matricea de pixeli ai unei imagini, dimensiunea imaginii si inca 2 perechi de coordonate ce vor reprezenta coltul din stanga sus si dreapta jos a portiunii pe care face zoom. Va trebui sa returneze un mesaj de tipul **MessageImage** care va contine o submatricea din matricea originala incadrata intre cele 2 puncte primite ca parametrul si noua dimensiune.

Imaginea rezultata va contine **ambele puncte** .
Ex: zoom=0,0,100,100 va avea o latime de 101 .

Flash

Componenta Flash va primi un mesaj de tipul MessageFlash ce va contine un camp de tip enum FlashType (ON, OFF, AUTO) si matricea de pixeli ai unei imagini precum si dimensiunea acesteia. In functie de optiunea utilizarii blitului, vom avea:

- ON - inseamna ca adaugam valoarea 50 la fiecare canal de culoare (RGB) al fiecarui pixel (cu mentiunea ca daca valoarea depaseste 255 sa o setam la 255)
- OFF - matricea ramane la fel
- AUTO
  - Se calculeaza luminozitatea medie a imaginii, calculand media aritmetica a luminozitatii tuturor pixelilor.
  - Formula pentru luminozitate este L = Math.round(0.2126*R + 0.7152*G + 0.0722*B)
  - Daca luminozitatea medie este mai mica decat 60 atunci vom adauga 50 la toate canalele (RGB) precum am facut la optiunea ON.
  - Altfel lasam imaginea asa cum este.

Va intoarce un mesaj de tipul **MessageImage** cu matricea de pixeli si dimensiunea ei, cu modificarile corespunzatoare optiunilor.

RawPhoto

Componenta Raw primeste un mesaj de tipul **MessageImage** si returneaza un mesaj tot de tipul**MessageImage** cu matricea rasturnata.
Best TODO: Implementati operatia de inversare in-place, utilizand O(1) memorie suplimentara.

NormalPhoto

Componenta **NormalPhoto** face acelasi lucru ca si **RawPhoto**, insa pentru a avea o simulara cat mai realista si pentru consistenta output-urilor generate, vom considera ca fiind o etapa separata si distincta de RawPhoto.

Totodata:

- Daca dorim sa facem o poza Raw in simulare, atunci va trece prin componenta RawPhoto care va intoarce imaginea.
- Daca dorim o poza normala, dupa etapa de precaptura, imaginea va trece mai intai prin componenta RawPhoto si apoi prin NormalPhoto fiind doua mesaje de tipuri diferite la baza (TaskType: RawPhoto si NormalPhoto). Motivatia ar fi:
  - In etapa de NormalPhoto, am putea complica transformarea, in sensul in care in loc de o simpla rasturnare sa faca o rotire a matricei de pixeli in functie de pozitia in care a fost facuta poza (Landscape sau Portrait) pentru a aparea natural atunci ne uitam la poza.
  - De asemenea, in implementare dorim sa vedem doua Componente distincte RawPhoto si NormalPhoto care se vor lega la MessageCenter-uri in functie de fisierul de configurare si sa reutilizati codul implementat.

Sepia

Componenta **Sepia** va primi un mesaj de tipul **MessageImage** si va returna tot un mesaj de tipul**MessageImage** ce vor contine matricele de pixeli si dimensiunea lor.
Va trebui sa schimbe valorile RGB pentru toti pixeli din imagine utilizand formulele urmatoare:

outputRed = Math.round((inputRed * 0.393) + (inputGreen * 0.769) + (inputBlue * 0.189));
outputGreen = Math.round((inputRed * 0.349) + (inputGreen * 0.686) + (inputBlue * 0.168));
outputBlue = Math.round((inputRed * 0.272) + (inputGreen * 0.534) + (inputBlue * 0.131));

Hint: Aveti grija la overflow (>255).

BlackWhite

Componenta **BlackWhite** va primi un mesaj de tipul **MessageImage** si va returna tot un mesaj de tipul**MessageImage** ce vor contine matricele de pixeli si dimensiunea lor.
Va trebui sa schimbe valorile RGB pentru toti pixeli din imagine utilizand formulele urmatoare:

outputRed = Math.round((inputRed * 0.3) + (inputGreen * 0.59) + (inputBlue * 0.11));
outputGreen = Math.round((inputRed * 0.3) + (inputGreen * 0.59) + (inputBlue * 0.11));
outputBlue = Math.round((inputRed * 0.3) + (inputGreen * 0.59) + (inputBlue * 0.11));

Blur

Componenta **Blur** va primi un mesaj de tipul **MessageImage** si va returna tot un mesaj de tipul**MessageImage** ce vor contine matricele de pixeli si dimensiunea lor.
Va trebui sa schimbe fiecare valoare din RGB cu media vecinilor (stanga-sus, sus, dreapta-sus, dreapta, dreapta-jos, jos, stanga-jos, stanga).
Se va calcula aceasta medie pe fiecare canal de culoare:

- outputRed = Math.round(suma_valorilor_de_rosu_a_vecinilor / numarul_de_vecini);
- outputGreen= Math.round(suma_valorilor_de_verde_a_vecinilor / numarul_de_vecini);
- outputBlue = Math.round(suma_valorilor_de_albastru_a_vecinilor / numarul_de_vecini);

Acest proces trebuie aplicat pentru fiecare pixel din imagine. Desigur pixelii din colturi, vor avea doar 3 vecini, pe cand cei care se afla pe prima linie, ultima linie, prima coloana, ultima coloana, dar nu in colturi, vor avea 5 vecini. Restul pixelilor vor avea 8 vecini.
Acest proces se va aplica de 10 ori pentru a observa usor efectul de blur in imagine, adica aplicarea a unui singur filtru blur inseamna a repeta procesul descris mai sus de 10 ori.

Workflow

Vom lua ca exemplu aceeasi schema de mai sus si vom descrie cum ar trebui sa functioneze aplicatia.

SimulationManager va contine o metoda care creaza reteaua de MessageCenter-uri pe baza **fisierului de configurare primit ca parametru** .
La fiecare centru de mesagerie, va da subscribe la componentele asociate centrului. De exemplu, pentru Madrid, va instantia o componenta de tip Flash si un de tip Zoom si le va da subscribe la centru de mesagerie. Apoi, conform listelor de vecini, va adauga vecinii corespunzatori fiecarui centru.
La un centru de mesagerie, se va folosi metoda publish de catre SimulationManager sau de catre un alt MessageCenter.

Apoi, SimulationManager va citi de la stdin comenzi cu formatul urmator:

input_image output_image pre(flash=[on|off|auto];zoom=sX,sY,eX,eY) photo(type=[raw,normal]) post(sepia;blur;black_white)

input_image va fi calea catre imaginea sursa (peisajul/contextul pe care dorim sa-l fotografiem) iar output_image va fi imaginea rezultat (fotografia in sine).
Parametru pre corespunde etapei de pre-captura, photo etapei de captura (raw fiind rasturanta si normal fiind normal) si post-captura cu cele trei filtre discutate mai sus.

Parametrii flash, type sunt mereu prezenti. Parametrul zoom impreuna cu toate filtrele pot sa lipseasca. De asemnea filtrele pot fi aplicate de ori cate ori pentru un efect mai intens.
Nu vor exista spatii in cadrul parametrilor pre, photo si post. Ex: pre(flash = on)
Separatorul dintre tipurile de filtre sau operatii aplicate este “;”.

SimulationManager se va opri cand va gasi comanda exit.

Exemple valide:

```
image1.jpg photo1.jpg pre(flash=on;zoom=0,0,100,100) photo(type=normal)
post(sepia;blur)

image2.jpg photo2.jpg pre(flash=off) photo(type=raw) post()

image3.jpg photo3.jpg pre(flash=auto;zoom=100,100,200,200) photo(type=normal)
post(black_white)

image4.jpg photo4.jpg pre(flash=on) photo(type=normal)
post(sepia;blur;blur;blur;sepia)

exit
```

Sa presupunem ca SimulationManager primeste urmatoarea comanda:

```
party.jpg selfie.jpg pre(flash=auto;zoom=100,100,300,400) photo(type=normal)
post(black_white;blur)
```

Primul task pe care trebuie sa-l rezolve SimulationManager este incarcarea imaginii in memorie. Pentru acest lucru, construieste un mesaj de tipul MessageLoad in care pune numele fisierului sursa “party.jpg” si seteaza atributul TaskType ca fiind ImageLoader. Trimite mesajul centrului de mesagerie Madrid pentru ca doar cu acesta comunica direct. Madrid se uita la tipul mesajului si la ce componente sunt inscrise la el. Avand doar Flash si Zoom, trebuie sa apeleze la vecini. Primul vecin este Paris si ii trimite acestuia mesajul apeland metoda publish al centrului respectiv de mesagerie. Paris are ca si componente subscribed doar RawPhoto si NormalPhoto. Neputand sa rezolva una din componentele sale task-ul din mesaj, trimite si mai departe catre vecini. Primul vecin din lista este Madrid. Il trimite centrului Madrid. Madrid nu poate sa-l ajute si trimite vecinilor. Analog mai sus, Madrid ii trimite mesajul centrului Paris.

Observam ca s-a creat o bucla infinita intre cele 2 centre. Pentru a rezolva aceasta situatie, fiecare mesaj va avea un atribut id de tip int care va fi unic pentru fiecare mesaj creat de SimulationManager. Astfel, logica de mai sus se modifica foarte putin, in sensul in care inainte ca un MessageCenter sa proceseze mesajul, verifica daca nu cumva mesajul cu acel id a mai trecut prin

acel nod.
Algoritmul pseudocod pentru metoda publish ar fi:

```
Message publish_algorithm(Message m):

  daca m.id este in lista id-urile procesate
  atunci
    returnez null;

  adaug id-ul mesajului in lista de id-uri ale mesajelor primite

    daca tasktype al mesajului m este egal cu tasktype-ul unei componente:
    atunci
        notific componenta ca are un mesaj nou
        si returnez mesajul rezultat de la componenta
    altfel
        pentru fiecare vecin din lista_de_vecini:
          trimit mesajul catre vecin

          daca mesajul primit de la vecin este diferit null
          atunci
            returnez mesajul venit de la vecin
          altfel
            voi continua cu urmatorul vecin

  //inseamna ca niciun vecin nu m-a putut ajuta sa rezolv task-ul din mesaj
  //asadar, voi intoarce null
  returnez null;
```

De mentionat ar fi ca metoda publish va afisa la consola numele centrului de fiecare data cand primeste un mesaj. Astfel, veti putea observa traseul pe care il va parcurge un mesaj in reteaua de MessageCenter-uri.

Urmarind noul algoritm in cazul de mai devreme in care aveam de rezolvat task-ul ImageLoader, avem urmatorii pasi:

- SimulationManager trimite mesaj MessageLoad catre Madrid.
- Acesta verifica daca id-ul mesajului se afla printre id-urile mesajelor primite deja.

- Fiind prima data cand il primeste, verifica apoi daca are o componenta de tipul ImageLoader.
- Neavand componenta corespunzatoare, trimite mesajul catre vecini.
- Trimite mesajul catre Paris, fiind primul vecin.
- Paris verifica daca mesajul a mai trecut pe la el.
- Nu a mai trecut si cum nu are componenta ImageLoader, trimite mesajul catre vecini.
- Primul vecin este Madrid caruia ii trimite mesajul.
- Madrid vazand ca a mai primit mesajul returneaza null.
- Paris trimite mesajul la urmatorul vecin Londra.
- Londra neavand ImageLoader, trimite mai departe catre primul vecin Bucuresti.
- Bucuresti nu are nici componenta ImageLoader si nici alti vecini. Returneaza null.
- Londra trimite la urmatorul vecin Berlin.
- Berlin are inscrisa o componenta de tipul ImageLoader si ii trimite mesajul primit.
- Berlin returneaza rezultatul de la componenta ImageLoader catre Londra.
- Londra returneaza rezultatul catre Paris.
- Paris returneaza rezultatul catre Madrid.
- Madrid returneaza rezultatul lui SimulationManager.

Nota: In consola va aparea urmatoarul sir de nume de centre:

```
Madrid
Paris
Madrid
Londra
Bucuresti
Berlin
```

Astfel, SimulationManager rezolva prima parte din comanda primita:

```
party.jpg selfie.jpg pre(flash=auto;zoom=100,100,300,400) photo(type=normal)
post(black_white;blur)
```

SimulationManager continua cu etapele din pre-captura: flash si zoom.
Prima data trimite un mesaj de tipul MessageFlash cu tasktype Flash punand in mesaj matricea primita de la ImageLoader si tipul de blit utilizat, in cazul curent fiind AUTO.
Centrul Madrid avand componenta Flash, ii trimite mesajul acesteia si rezultatul va fi intors catre SimulationManager.
Analog va proceda si cu Zoom, trimitand un mesaj de tip MessageZoom cu matricea de pixeli anterioara si cu parametrii de la Zoom (cele 2 perechi de coordonate).

Urmatoare etapa este cea de captura in care avem type=normal. Asta inseamna ca matricea de pixeli va trece mai intai prin componenta RawPhoto si apoi prin componenta NormalPhoto.

SimulationManager creaza un mesaj de tip MessageImage cu tipul RawPhoto si il trimite lui Madrid care va trimite mai departe MessageCenter-ului Paris. Acesta trimite componentei RawPhoto si intoarce rezultatul.
SimulationManager creaza in continuare un alt mesaj de tipul MessageImage cu tipul NormalPhoto si il trimite lui Madrid, ajungand in final tot la Paris care contine componenta NormalPhoto.

Ultima etapa este cea de post-captura in care putem aplica mai multe filtre.
SimulationManager, in acest caz, va trebui sa aplice filtrul black_white si blur.
Construieste mesajele corespunzatoare si le trimite lui Madrid care mai departe va folosi vecinii pentru a rezolva task-urile dupa algoritmul descris anterior.

Ultimul lucru pe care trebuie sa-l faca SimulationManager este de a salva fotografia in fisierul destinatie.
Va crea un mesaj de tipul MessageSave cu matricea de pixeli rezultata dupa filtrul de blur si va ajunge ca in cazul initial la centrul Berlin unde ImageSaver-ul o va salva in fisier.

In continuare, SimulationManager va citi urmatoarele comenzi si le va executa asemanator pana cand va primi comanda exit.

Alte informatii

Trebuie:

- Metoda main sa se afle in SimulationManager.java iar acesta sa fie in directorul src/.
- Sa creati mai intai toate MessageCenter-urile cu componentele corespunzatoare si apoi sa populati lista de vecini cu instantele corespunzatoare.
- Sa memorati in MessageCenter-uri toate id-urile mesajelor primite pentru a evita ciclarea mesajelor in retea.
- Sa utilizati Math.round in formule pentru a avea acelasi output cu testele.
- Sa aveti grija la valorile peste 255 sau mai mici ca 0.
- Sa va construiti si celalte tipuri de mesaje: MessageZoom si MessageFlash.

Puteti:

- Adauga oricate clase auxiliare si oricate pachete.
- Folosi o clasa unde sa va stocati constante de tip String sau Integer.

Schelet de cod & Testare

- **Updated:** Schelet de cod
- Componentele ImageLoad si ImageSave **updated**: ImageLoad_ImageSave.zip
- Checker
- Extra #Selfie

Pentru a testa tema, copiati continutul arhivei checker.zip in directorul proiectului eclipse sau directorul src/ in directorul checker-ului si apoi rulati script-ul checker.sh

In caz ca vor exista update-uri pentru checker, acesta verifica la inceput daca aveti ultima versiune de checker.

Pentru a rula checker-ul pe windows, puteti sa:

- instalati git bash : http://git-scm.com/downloads
- sa adaugati in variabila de mediu PATH calea catre directorul bin din jdk:https://www.youtube.com/watch?v=P6NeRfcJtdU
- din terminalul bash instalat, rulati scriptul checker.sh