Domaine Sciences et Technologies
Master Bio-informatique : DLAD

Aix*Marseille université Socialement engagée

Programmation avancée : TP 6 bonus
Code UE : SBICU05

Année 2023-24

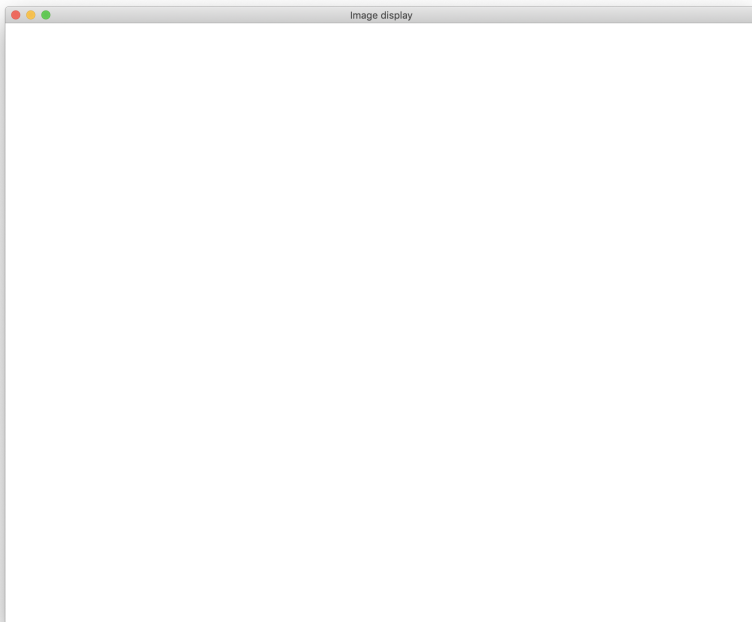
Images

Images

Consignes pour démarrer le TP

Comme pour le TP 2, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du TP 2. Le lien vers le projet à forker est le suivant : https://etulab.univ-amu.fr/alaboure/color-image-template.

Une fois le dépôt téléchargé, vous pouvez compiler et exécuter le code cliquer deux fois sur `color-image` -> `application` -> `run`. Vous devriez obtenir l'affichage suivant.



Pour exécuter les tests, il faut passer par l'onglet gradle à droite et cliquer deux fois sur `color-image` -> `Tasks` -> `verification` -> `test`.

Représentations d'images en couleurs

On va considérer quatre manières de représenter une image en couleur et donc quatre classes d'images :

— `BruteRasterImage` : dans cette représentation, on stocke pour chaque pixel sa couleur sous la forme d'un objet `Color`. Une image de ce type contiendra donc une matrice `Color[][] pixels`.
— `PaletteRasterImage` : dans cette représentation, on stocke :

  - une palette consistant en une liste de Color qui correspondent aux couleurs utilisées dans l'image,
  - pour chaque pixel l'indice de sa couleur dans la palette encodé avec un int.

  Cette manière de faire a l'avantage d'économiser de la mémoire pour les images ayant peu de couleurs (un byte prenant beaucoup moins de place en mémoire qu'une Color). Une image de ce type contiendra donc une liste List<Color> palette et une matrice int[][] indexesOfColors.
- SparseRasterImage : dans cette représentation, on stocke les coordonnées et la couleur de chacun des pixels de l'image qui n'est pas blanc (Color.WHITE). Une image de type contiendra un dictionnaire Map<Point,Color> pixelsMap (il est expliqué dans la suite du sujet à quoi sert un dictionnaire).
- VectorImage : dans cette représentation, on stocke une liste de formes géométriques (utilisant l'interface Shape). Une image de type sera construite à partir de List<Shape> shapes. Chaque forme géométrique (qui peut être un cercle, un rectangle, ...) contient un certain nombre de points et définit la couleur des points à l'intérieur. La couleur d'un point de l'image est la couleur de la première forme dans la liste qui contient le point ou bien blanc si le point n'est contenu dans aucune forme.

Gestion de versions

Comme pour les TP précédents, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du précédent TP.

Consignes pour le début du TP

Modifiez le fichier README.md. Mettez votre nom, votre **numéro de groupe** ainsi que le nom et le **numéro de groupe** de votre éventuel co-équipier. Faites un commit avec pour message "inscription d'un membre de l'équipe", puis un push.

Tâche 1

Classe BruteRasterImage

Écrivez le code de la classe BruteRasterImage qui implémente l'interface Image. La classe devra avoir :

- deux constructeurs :
  - public BruteRasterImage(Color color, int width, int height) : construisant une image de la taille spécifiée et dont tous les pixels sont de la couleur spécifiée.
  - public BruteRasterImage(Color[][] colors) : construisant une image à partir de la matrice donnée en paramètre. Le premier indice correspondant à la coordonnée en x et le deuxième indice correspondant à la coordonnée en y. Il vous faudra vous assurer que la matrice de couleurs données en paramètre soit correcte. Vous pouvez utiliser pour cela les méthodes de la classe Matrices contenue dans le paquet util.
- neufs méthodes :
  - public void initializeRepresentation() : alloue la matrice représentant l'image (à utiliser dans le constructeur).
  - public void setPixelColor(Color color, int x, int y) : fixe la couleur d'un pixel.
  - public Color getPixelColor(int x, int y) : retourne la couleur d'un pixel.

— `private void setPixelsColor(Color[][] pixels)` : met à jour les valeurs de couleurs de l'image en utilisant les valeurs de la matrice donnée en paramètre.
— `private void setPixelsColor(Color color)` : change les valeurs de tous les pixels pour qu'ils soient tous de la couleur donnée en paramètre.
— `public int getWidth()` : retourne la largeur de l'image.
— `public int getHeight()` : retourne la hauteur de l'image.
— `protected void setWidth(int width)` : fixe la largeur de l'image.
— `protected void setHeight(int height)` : fixe la hauteur de l'image.

Test de la classe `BruteRasterImage`

Afin de tester la classe, vous allez :

— Créer une énumération nommée `RasterImageType` qui contiendra pour le moment qu'une valeur constante nommée `BRUTE`. Cette énumération nous servira à spécifier un type d'image et vous rajouterez les deux autres types d'images *raster* par la suite.
— Créer une classe `NotSupportedException` qui étend `RuntimeException` et qui a un constructeur prenant un message de type `String` en paramètre.
— Ajouter au paquet `image` la classe `RasterUniformImageFactory` suivante définie par le fichier RasterUniformImageFactory.java.
— Ajouter dans la méthode `initialize` de la classe `Display` (dans le paquet `viewer`) une affectation de l'attribut `imageFactory` avec l'instruction `imageFactory = new RasterUniformImageFactory(200, 200, Color.RED, RasterImageType.BRUTE;`. Vou devriez obtenir l'affichage d'une fenêtre rouge de 200 pixels sur 200 pixels.
— Ajouter au paquet `image` la classe `RasterFlagFactory` suivante définie par le fichier RasterFlagFactory.java.
— Changer dans la méthode `initialize` de la classe `Display` (dans le paquet `viewer`) l'affectation de l'attribut `imageFactory` en mettant l'instruction `imageFactory = new RasterFlagFactory(900, 600, Color.BLUE, Color.WHITE, Color.RED, RasterImageType.BRUTE);`. Vous devriez obtenir l'affichage du drapeau Français.

Tâche 2

Classe `PaletteRasterImage`

Écrivez le code de la classe `PaletteRasterImage` qui implémente l'interface `Image`. La classe devra avoir :

— deux constructeurs :
  — `public PaletteRasterImage(Color color, int width, int height)` : construisant une image de la taille spécifiée et dont tous les pixels sont de la couleur spécifiée .
  — `public PaletteRasterImage(Color[][] pixels)` : construisant une image à partir de la matrice donnée en paramètre. Le premier indice correspondant à la coordonnée en x et le deuxième indice correspondant à la coordonnée en y.
— neufs méthodes :

- `public void initializeRepresentation()` : alloue la liste pour stocker la palette et la matrice représentant l'image (à utiliser dans le constructeur).
- `public void setPixelColor(Color color, int x, int y)` : fixe la couleur d'un pixel (en ajoutant la couleur à la palette si elle n'était pas dans la palette).
- `public Color getPixelColor(int x, int y)` : retourne la couleur d'un pixel.
- `public void setPixelsColor(Color[][] pixels)` : met à jour les valeurs de couleurs de l'image en utilisant les valeurs de la matrice donnée en paramètre.
- `private void setPixelsColor(Color color)` : change les valeurs de tous les pixels pour qu'ils soient tous de la couleur donnée en paramètre.
- `public int getWidth()` : retourne la largeur de l'image.
- `public int getHeight()` : retourne la hauteur de l'image.
- `protected void setWidth(int width)` : fixe la largeur de l'image.
- `protected void setHeight(int height)` : fixe la hauteur de l'image.

Test de la classe `PaletteRasterImage`

Pour tester la classe `PaletteRasterImage`, vous allez :

- Ajouter la valeur `PALETTE` à l'énumération `RasterImageType`.
- Changer le code présent dans les classes `RasterFlagFactory` et `RasterUniformImageFactory` pour créer des images de type `PaletteRasterImage` dans la méthode `makeImage` lorsque l'attribut `rasterImageType` est égal à `PALETTE`
- Changer le code présent dans `Display` pour construire des images du bon type.

Tâche 3

Factorisation du code

Les classes `BruteRasterImage` et `PaletteRasterImage` ont beaucoup de code en commun. Comme nous l'avons vu dans le cours, la duplication de code est quelque chose qu'un bon programmeur essaye d'éviter.

Créez un classe abstraite `RasterImage` qui sera étendue par `BruteRasterImage` et `PaletteRasterImage` afin d'éviter la duplication de code.

Tests

Vous devez vous assurez que vous n'avez pas changé le comportement des deux classes `BruteRasterImage` et `PaletteRasterImage` en factorisant le code.

Tâche 4

Classe `SparseRasterImage`

Écrivez le code de la classe `SparseRasterImage` qui étend `RasterImage`.

Afin de stocker l'association entre les couleurs et les pixels dans `SparseRasterImage`, vous allez utiliser l'interface `Map<K,V>`. Cette interface, qui est implémentée (en autre) par la classe `HashMap<K,V>`, permet d'associer des clés (de type `K`) à des valeurs (de type `V`). Elle contient (entre autre) les méthodes suivantes :

- `boolean containsKey(Object key)`: Returns `true` if this map contains a mapping for the specified `key`.
- `V get(Object key)`: Returns the value to which the specified `key` is mapped, or `null` if this map contains no mapping for the `key`.
- `V getOrDefault(Object key, V defaultValue)`: Returns the value to which the specified `key` is mapped, or `defaultValue` if this map contains no mapping for the `key`.
- `V put(K key, V value)`: Associates (maps) the specified `value` with the specified `key` in this map.

Vous devez associer des `Point` à des `Color`.

La classe devra avoir :

- deux constructeurs :
  - `public SparseRasterImage(Color color, int width, int height)` : construisant une image de la taille spécifiée et dont tous les pixels sont de la couleur spécifiée.
  - `public SparseRasterImage(Color[][] pixels)` : construisant une image à partir de la matrice donnée en paramètre. Le premier indice correspondant à la coordonnée en x et le deuxième indice correspondant à la coordonnée en y.
- neufs méthodes :
  - `public void initializeRepresentation()` : crée le dictionnaire `HashMap<Point,Color>` pour stocker l'association entre points et couleurs (à utiliser dans le constructeur).
  - `public void setPixelColor(Color color, int x, int y)` : fixe la couleur d'un pixel (en associant le point de coordonnée (x, y) à la couleur).
  - `public Color getPixelColor(int x, int y)` : retourne la couleur d'un pixel.
  - `private void setPixelsColor(Color[][] pixels)` : met à jour les valeurs de couleurs de l'image en utilisant les valeurs de la matrice donnée en paramètre.
  - `private void setPixelsColor(Color color)` : change les valeurs de tous les pixels pour qu'ils soient tous de la couleur donnée en paramètre.
  - `public int getWidth()` : retourne la largeur de l'image.
  - `public int getHeight()` : retourne la hauteur de l'image.
  - `protected void setWidth(int width)` : fixe la largeur de l'image.
  - `protected void setHeight(int height)` : fixe la hauteur de l'image.

Test de la classe `SparseRasterImage`

Pour tester la classe `SparseRasterImage`, vous allez :

- Ajouter la valeur `SPARSE` à l'énumération `RasterImageType`.
- Changer le code présent dans les classes `RasterFlagFactory` et `RasterUniformImageFactory` pour créer des images de type `SparseRasterImage` dans la méthode `makeImage` lorsque l'attribut `rasterImageType` est égal à `Sparse`

— Changer le code présent dans `Display` pour construire des images du bon type.

Tâche 5

Classe `Rectangle`

Écrivez une classe `Rectangle` qui implémente l'interface `Shape`.

La classe devra avoir :

— un constructeur : `Rectangle(int x, int y, int width, int height, Color color)` : qui construit un rectangle dont le coin en haut à gauche a pour coordonnées (x, y) avec les largeur, hauteur et couleur spécifiées (dans une interface graphique l'origine du système de coordonnées est le coin en haut à gauche, l'axe des x va de la gauche vers la droite et l'axe des y va de haut en bas).

— deux méthodes :

  — `public boolean contains(Point point)` : qui renvoie `true` si le point est à l'intérieur du rectangle (bord inclus) et `false` sinon.

  — `public Color getColor()` : qui renvoie la couleur associé au rectangle.

Classe `VectorImage`

Écrivez le code de la classe `VectorImage` qui implémente `Image`.

La classe devra avoir :

— un constructeur :

  — `VectorImage(List<Shape> shapes, int width, int height)` : construisant une image de la taille spécifiée et avec les formes données en paramètre.

— cinq méthodes :

  — `public Color getPixelColor(int x, int y)` : retourne la couleur d'un pixel.

  — `public int getWidth()` : retourne la largeur de l'image.

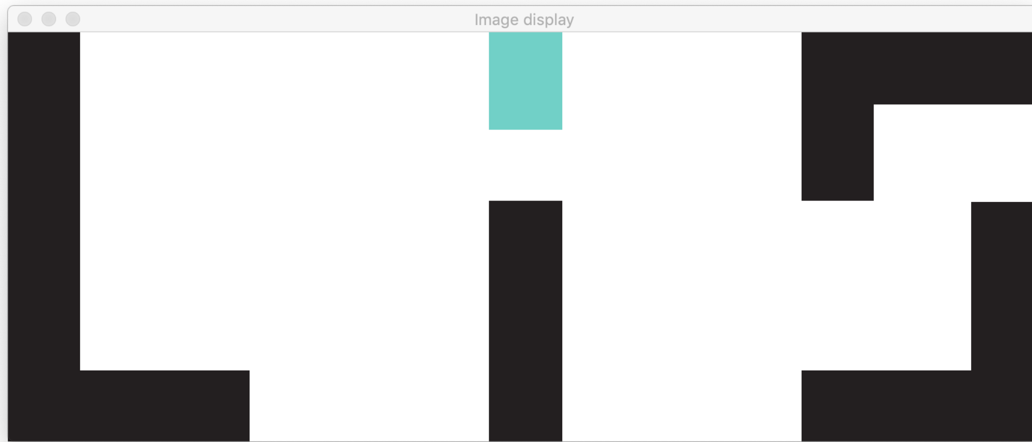
  — `public int getHeight()` : retourne la hauteur de l'image.

  — `protected int setWidth(int width)` : fixe la largeur de l'image.

  — `protected int setHeight(int height)` : fixe la hauteur de l'image.

Test de la classe `VectorImage`

Pour tester la classe `VectorImage`, ajouter au paquet `image` la classe `LogoLISFactory` définie par le fichier LogoLISFactory.java. Puis changer le code présent dans `Display` pour utiliser cette nouvelle classe. Vous devriez obtenir l'affichage de l'image ci-dessous.



Tâches supplémentaires

— Rajouter de nouvelles implémentations de l'interface `Shape` pour faire des cercles, des triangles rectangles, des triangles quelconques, des polygones quelconques, ...

— Réduire la duplication de code entre `RasterImage` et `VectorImage`.

— Écrire des tests et de la documentation pour toutes vos classes.