RAPPORT SAE 3.2 – Primitive Image Format

Par Valentin LOISON et Dimitri SOLAR

Date : Janvier 2026

Sommaire :

Partie 1 : Introduction au projet PIF

L'objectif de ce projet est de concevoir une application capable de compresser et de visualiser des images au format **PIF** (Primitive Image Format). Ce format vise à résoudre les problèmes de taille des formats simplifiés en utilisant une compression sans perte basée sur l'algorithme de **Huffman**. Le principe est d'attribuer des codes binaires courts aux couleurs les plus fréquentes et des codes plus longs aux couleurs rares, réduisant ainsi le poids total du fichier sans aucune altération de l'image originale.

Partie 2 : Description des fonctionnalités

Cette section détaille les capacités opérationnelles des deux programmes développés pour la manipulation du format PIF. L'accent a été mis sur l'ergonomie et la robustesse de l'affichage.

2.1 Le Visualisateur

Le visualisateur est l'outil destiné à la consommation des images compressées. Son rôle est de rendre transparente la complexité du décodage de Huffman pour l'utilisateur final.

Gestion de l'ouverture : Le programme accepte un chemin de fichier directement en argument de la ligne de commande. En l'absence d'argument, il invoque un JFileChooser filtré pour ne proposer que des extensions .pif, garantissant une expérience utilisateur fluide.

Affichage adaptatif : Lors du chargement, la fenêtre calcule les dimensions optimales en fonction de la résolution de l'image (extraite de l'en-tête du fichier) tout en s'assurant de ne pas déborder des limites de l'écran de l'utilisateur.

Système de Panoramique (Pan) : Pour les images haute résolution dépassant la taille de la fenêtre :

- **Interaction :** En maintenant le bouton gauche de la souris enfoncé, l'utilisateur peut faire glisser l'image dans toutes les directions.
- **Contraintes de bordures :** Le PanneauImage calcule en temps réel les décalages (decalageX, decalageY) pour empêcher l'image de sortir du cadre visible, assurant ainsi que les bords de l'image restent toujours collés aux bords de la fenêtre si l'utilisateur tente d'aller trop loin.

Centrage intelligent : Si l'image est plus petite que la zone d'affichage, la méthode paintComponent du PanneauImage la centre automatiquement pour un meilleur confort visuel.

2.2 Le Convertisseur

Le convertisseur est l'outil technique permettant de passer d'un format standard (PNG, JPG, BMP) au format PIF via une analyse statistique.

Analyse Multi-Canaux : Dès le chargement d'une image via ImageIO, le programme sépare les données en trois tableaux distincts : Rouge, Vert et Bleu.

Interface Statistique (Onglets) : Une fenêtre à onglets (JTabbedPane) présente à l'utilisateur une analyse détaillée pour chaque canal :

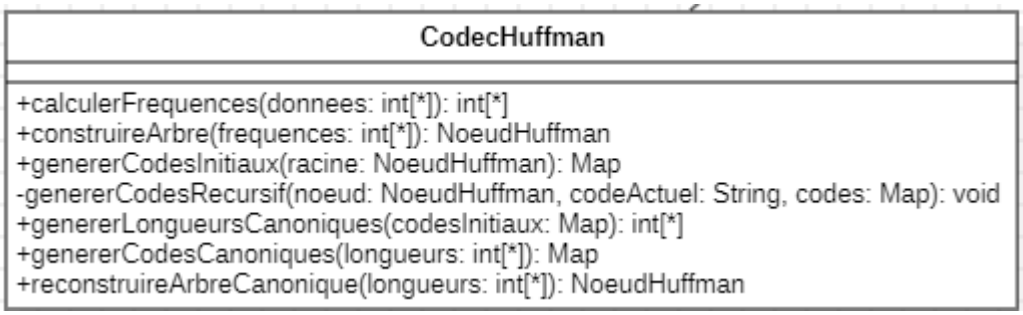
- **Table des fréquences :** Nombre d'occurrences de chaque valeur de pixel (0-255).
- **Codes Initiaux :** Les codes générés par l'arbre de Huffman classique.
- **Codes Canoniques :** Les codes finaux après tri par longueur, tels qu'ils seront inscrits dans le fichier.

Aperçu et Conversion : Une miniature de l'image source est affichée pour confirmer le chargement. L'utilisateur déclenche ensuite la conversion via ActionConvertir, qui génère le fichier .pif incluant les 768 octets de tables de longueurs et le flux binaire compressé.

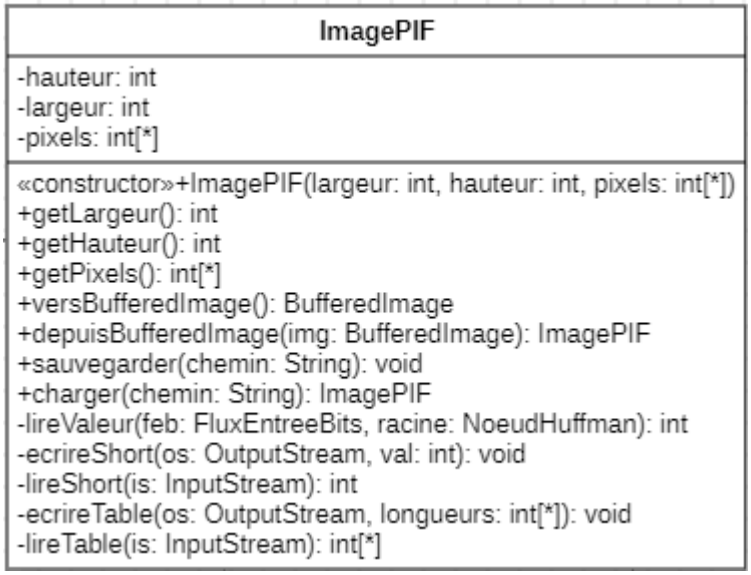
Partie 3 : Structure du programme

L'application suit une architecture modulaire pour séparer l'interface de la logique de compression :

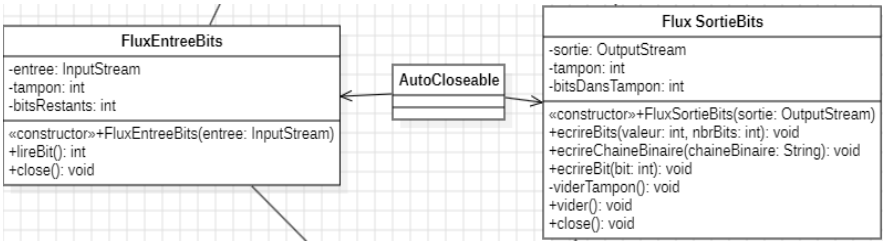
Moteur Algorithmique : CodecHuffman gère la logique de l'arbre et des codes canoniques.



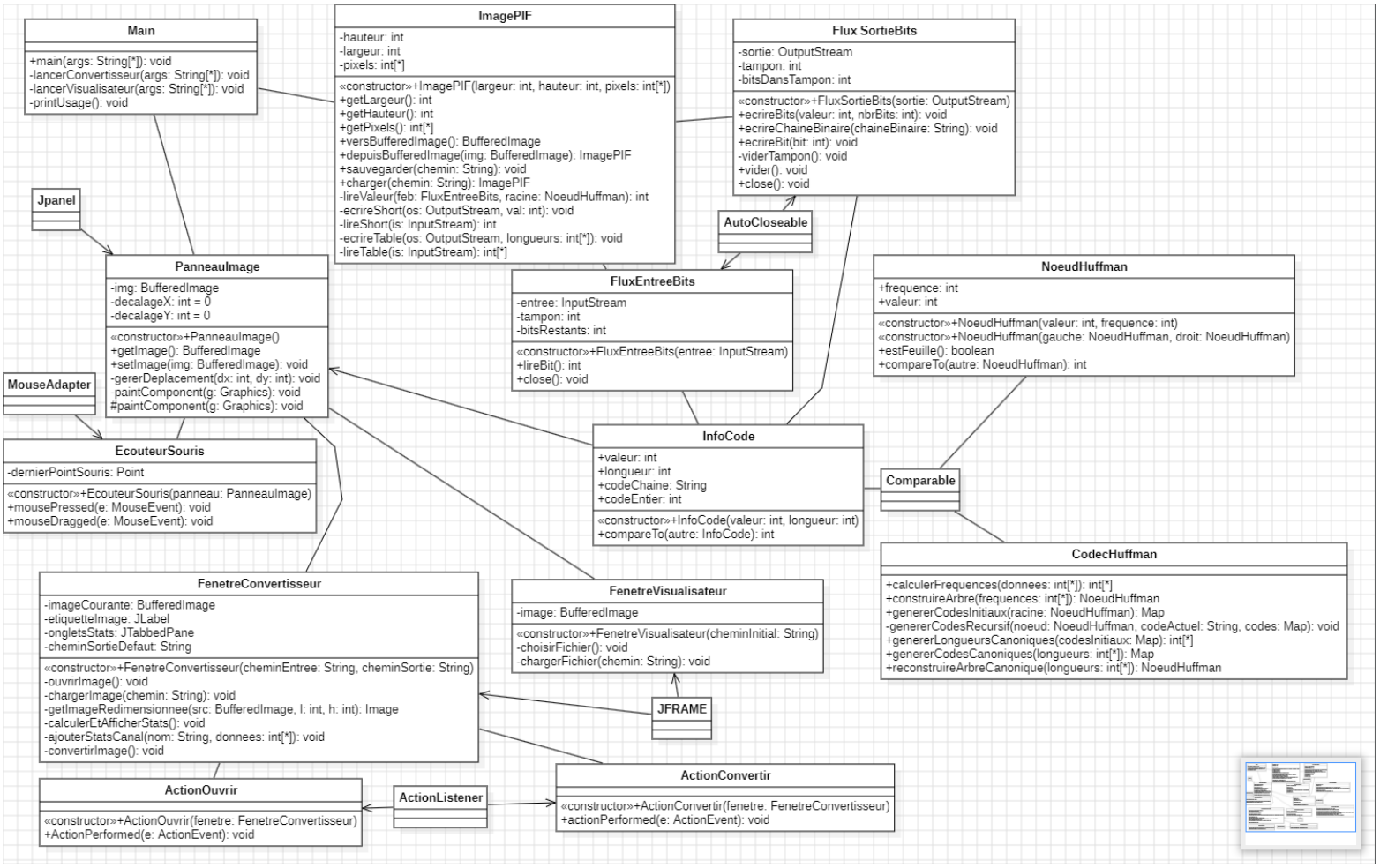
Modèle Image : ImagePIF encapsule les données binaires et les méthodes de sauvegarde/chargement.



Entrées/Sorties : FluxEntreeBits et FluxSortieBits assurent la lecture et l'écriture au bit près.



Pour à la fin avoir un diagramme de classes complet



Partie 4 : Étude de l'algorithme Huffman et Arbre Binaire

Cette partie détaille la logique mathématique et informatique permettant la compression sans perte du format PIF.

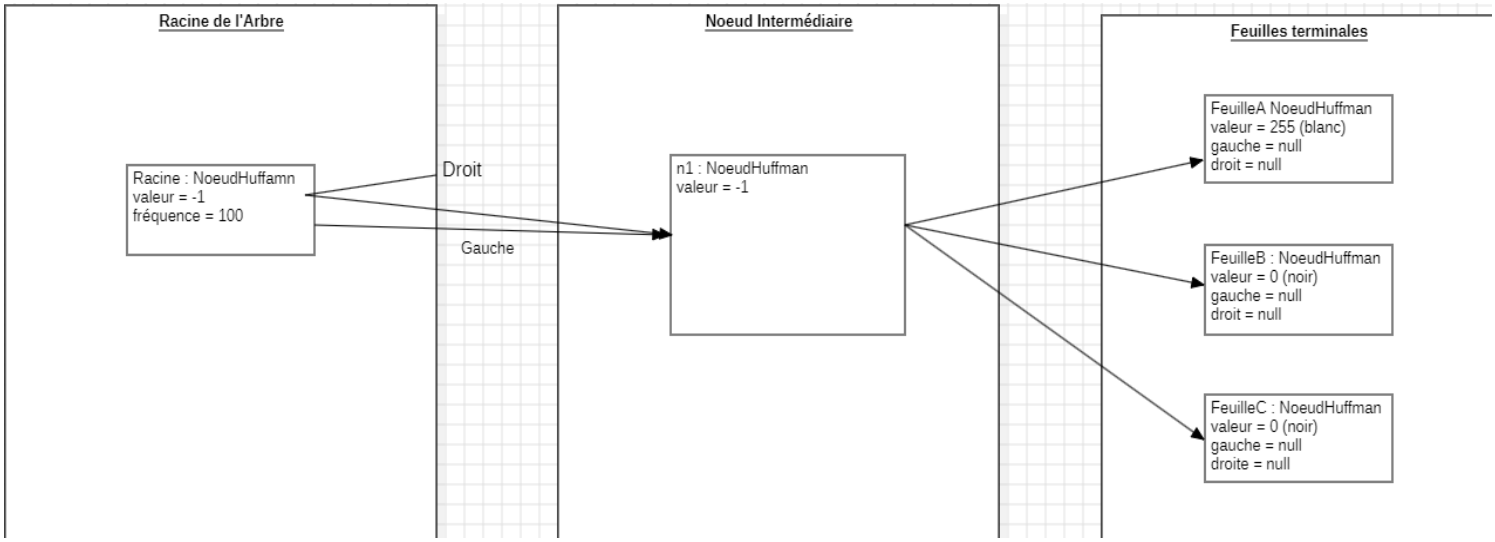
4.1 L’arbre binaire de Huffman

La compression repose sur la création d'un arbre binaire optimal pour chaque canal de couleur (Rouge, Vert, Bleu).

Calcul des fréquences : Avant de construire l'arbre, le programme scanne l'intégralité des pixels pour compter le nombre d'occurrences de chaque valeur (de 0 à 255). Plus une couleur est présente, plus sa priorité est haute.

Algorithme de construction : 1. Chaque valeur de pixel devient un NoeudHuffman de type "feuille". 2. Ces nœuds sont insérés dans une **file de priorité** (PriorityQueue), ordonnée par fréquence croissante. 3. Tant qu'il reste plus d'un nœud dans la file, le programme extrait les deux nœuds ayant les plus petites fréquences et les fusionne en un nouveau nœud parent. La fréquence du parent est la somme de celle de ses enfants. 4. Ce processus se répète jusqu'à ce qu'il ne reste qu'un seul nœud : la **racine** de l'arbre.

4.2 Diagramme Explication Huffman canonique (Structure de l'arbre)



L'arbre ainsi généré possède une structure hiérarchique où chaque embranchement représente un choix binaire (0 pour la gauche, 1 pour la droite).

Nœuds internes : Ils ne contiennent pas de pixel (valeur = -1) mais servent de pivots de direction.

Feuilles : Elles contiennent la valeur réelle du pixel et marquent la fin d'un code.

Cela montre comment les objets NoeudHuffman sont liés. On remarque que les valeurs les plus fréquentes se trouvent plus proches de la racine, réduisant ainsi la longueur de leur chemin binaire.

4.3 Le passage au Codage Canonique

Une fois l'arbre de Huffman standard généré, nous ne l'utilisons pas tel quel pour le stockage. Nous le transformons en **code canonique**.

Pourquoi le canonique ? Stocker un arbre complet dans un fichier prendrait trop de place. Le code canonique permet de reconstruire l'arbre

identique uniquement à partir de la **longueur des codes** de chaque symbole.

Règle de construction : Les codes sont attribués en triant les symboles d'abord par la longueur de leur code (croissant), puis par leur valeur numérique (0 à 255).

Avantage : Dans le fichier PIF, il nous suffit de réserver 256 octets par canal (soit 768 octets au total) pour stocker uniquement ces longueurs. À la lecture, la méthode reconstruireArbreCanonique recrée l'arbre complet sans erreur.

Partie 5 : Mécanismes d'encodage et décodage

Cette partie expose les processus techniques permettant de transformer une image pixellisée en un flux binaire compressé et inversement.

5.1 Le processus d'encodage (Convertisseur)

La conversion repose sur une transformation rigoureuse des données colorimétriques en séquences binaires optimisées.

Extraction des composantes : L'image source est décomposée en trois canaux (Rouge, Vert, Bleu). Chaque canal fait l'objet d'un codage de Huffman indépendant pour maximiser l'efficacité de la compression selon la répartition des couleurs.

Génération des tables : Pour chaque canal, le programme génère une table de codes canoniques via CodecHuffman.genererCodesCanoniques. On ne conserve que la longueur de chaque code (en bits) pour l'inscrire dans l'en-tête du fichier.

Écriture du flux : Pour chaque pixel de l'image, le convertisseur récupère le code binaire associé à sa valeur de couleur et l'envoie vers le FluxSortieBits.

5.2 Le processus de décodage (Visualisateur)

Le décodage est l'opération inverse qui doit reconstruire l'image originale sans aucune perte d'information.

Reconstruction de l'arbre : À partir des 768 octets de tables (256 par canal), le visualisateur utilise reconstruireArbreCanonique pour recréer l'arbre binaire de Huffman. Comme les codes sont canoniques, la seule connaissance de leur longueur suffit à rebâtir un arbre identique à celui utilisé lors de l'encodage.

Navigation binaire : Le programme lit les bits un par un dans le fichier. Pour chaque canal :

- Si le bit est 0, il descend vers l'enfant **gauche** du nœud actuel.
- Si le bit est 1, il descend vers l'enfant **droit**.

Récupération de la couleur : Dès qu'une "feuille" de l'arbre est atteinte, la valeur du pixel est extraite et le parcours recommence depuis la racine pour le canal suivant.

5.3 Gestion technique des flux : Manipulation bit à bit

Comme la classe standard OutputStream de Java ne permet d'écrire que des octets entiers (8 bits), nous avons dû implémenter des classes spécifiques pour gérer les codes de longueurs variables (ex: un code de 3 bits).

Classe FluxSortieBits (Le tampon) : Cette classe utilise un "tampon" (buffer) entier. Chaque nouveau bit est ajouté à ce tampon via des opérations de décalage vers la gauche (<<) et un "OU" binaire (|). Dès que le tampon contient 8 bits, la méthode viderTampon écrit l'octet complet dans le fichier via sortie.write().

Classe FluxEntreeBits : À l'inverse, cette classe lit un octet complet depuis le disque mais ne distribue les bits à l'application qu'un par un. Elle gère un compteur bitsRestants pour savoir quand appeler entree.read() pour remplir à nouveau son tampon interne.

Partie 5 : Conclusion personnels

Valentin LOISON :

Intégrer ce projet en binôme avec Dimitri a été une expérience très formatrice, notamment sur l'aspect "Interface Homme-Machine" (IHM) et la manipulation d'objets graphiques complexes en Java.

Ma mission s'est concentrée sur le rendu visuel et l'interactivité du visualisateur. Le développement du PanneauImage a été particulièrement intéressant : il a fallu gérer mathématiquement le centrage des petites images et, surtout, le système de déplacement (panoramique) pour les images dépassant de l'écran. La synchronisation entre l' EcouteurSouris et le rafraîchissement du panneau graphique m'a permis de mieux maîtriser le fonctionnement interne de la bibliothèque Swing.

Au-delà de l'aspect graphique, comprendre comment transformer un flux de bits "bruts" en une BufferedImage affichable a été un véritable saut de compétences. Ce projet m'a montré que la qualité d'une application dépend autant de la puissance de son algorithme de compression que de la fluidité de son interface utilisateur.

Dimitri SOLAR :

Le projet de développement du format PIF a représenté un défi technique bien plus axé sur l'optimisation algorithmique. Ma principale contribution a porté sur la logique de compression et la gestion des flux binaires.

La plus grande difficulté a été l'implémentation de la classe FluxSortieBits car une erreur d'un seul bit décale l'intégralité du fichier et rend l'image illisible. J'ai également beaucoup appris sur la reconstruction d'arbres à partir de codes canoniques, ce qui permet de comprendre comment des standards comme le JPEG parviennent à une telle efficacité.

Sur le plan organisationnel, la collaboration avec Valentin a été très productive. Nous avons utilisé StarUML pour valider notre architecture avant de coder, ce qui nous a permis de diviser le travail efficacement : pendant que je me concentrais sur le moteur de compression (CodecHuffman), Valentin pouvait avancer sur les problématiques d'affichage et d'interface.