SAé Semestre 3 - Primitive Image Format

Canpolat DEMIRCI–ÖZMEN, Maxime ELIOT et Luka PLOUVIER
BUT2 Informatique (FI) | 2025/2026 | UPEC - IUT de Fontainebleau

Sommaire :

I. Introduction

Lors de cette seconde SAé du semestre 3, nous avons été amenés à concevoir et implémenter un format d'image simplifié, nommé Primitive Image Format (PIF), ainsi que les outils logiciels nécessaires à sa création et à sa visualisation. Ce projet s'inscrit dans la continuité des enseignements portant sur la manipulation des fichiers binaires, les structures de données (notamment les arbres) et les algorithmes fondamentaux de compression.

Le format PIF repose sur une séparation des **composantes** de couleur **rouge**, **verte** et **bleue**, qui sont analysées de manière indépendante. Cette analyse permet de calculer la fréquence d'apparition de chaque valeur de couleur de l'image. Ces fréquences sont ensuite utilisées pour appliquer le **codage** de **Huffman** (sous sa forme **canonique**), un algorithme de compression sans perte, sous la forme d'un **arbre**, qui attribue des codes binaires de longueur variable selon la fréquence des valeurs. Les valeurs les plus fréquentes sont codées sur moins de bits, ce qui permet de réduire la taille globale du fichier.

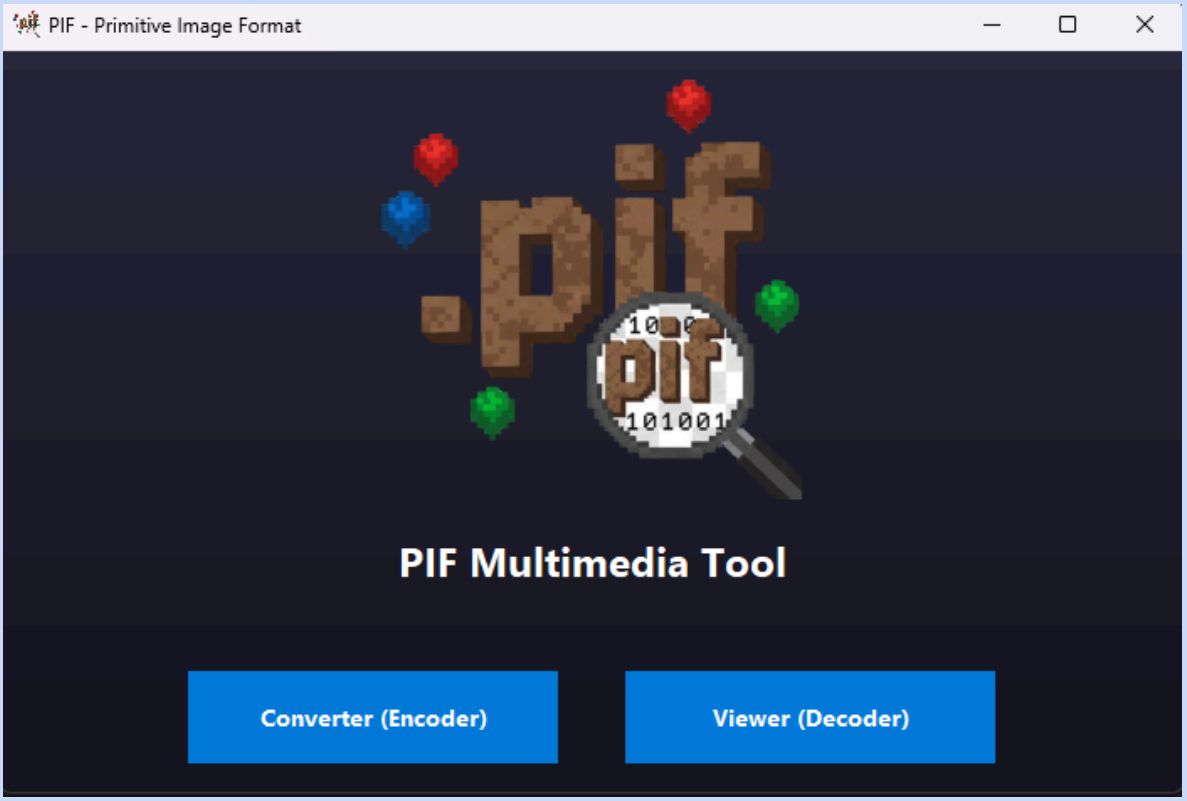
Afin de rendre ce format réellement utilisable, deux applications distinctes ont été développées en langage Java en suivant l'architecture **MVC**.
La première est un **convertisseur**, dont le rôle est d'ouvrir une image classique, d'analyser ses pixels, puis de générer un fichier au format PIF en gérant l'écriture bit à bit.
La seconde est un **visualisateur**, permettant de lire un fichier PIF, de reconstruire les arbres de décodage et d'afficher l'image. Une attention particulière a été portée à l'ergonomie de l'interface graphique.

La réalisation de ce projet nous a permis de mettre en pratique plusieurs notions vues au cours du semestre, telles que les arbres binaires, les files de priorité, la programmation orientée objet et la manipulation fine des flux binaires. Elle nous a également permis de mieux comprendre le fonctionnement interne des formats d'image ainsi que les compromis existants entre simplicité, performance et taux de compression.

II. Description des fonctionnalités du Primitive Image Format

A. Écran d'accueil

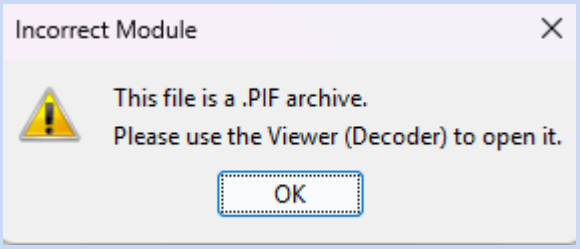
Si l'utilisateur lance l'application sans avoir donné de nom fichier en argument, une fenêtre d'accueil apparaît. Cette dernière le laisse choisir entre les deux logiciels développés : le convertisseur ou le visualisateur. Les deux ouvrent par conséquent un JFileChooser.
On peut également passer des **arguments** à notre **Main** : si un **fichier .pif** est passé en argument, on démarre le **visualiseur** (décodeur). Si un fichier **image** est passé en argument, on démarre le **convertisseur**. Et si, en plus du fichier image, le nom voulu pour le fichier .pif est passé en **2e argument**, le convertisseur se lance et **convertit automatiquement** le fichier image en PIF.

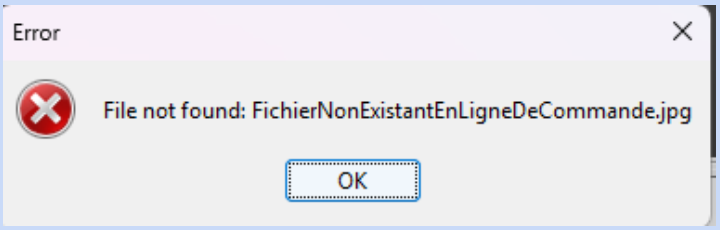


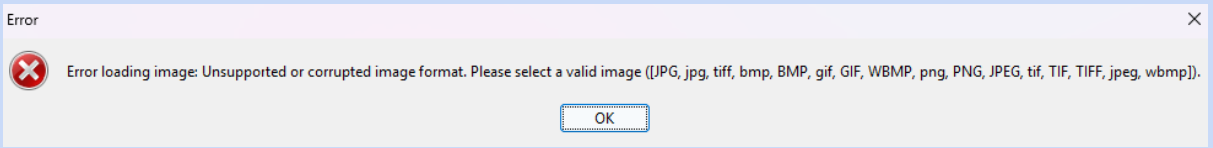
On peut également démarrer le convertisseur ou encore le visualisateur de façon indépendante.
Si aucun argument est passé, la fenêtre s'ouvre.
Pour le **convertisseur** :
java -jar Converter.jar res/icons/icon.png res/icon.pif
ou encore *make run-conv ARGS="res/icon.png res/resultat.pif"*
Pour le **visualisateur** :
java -jar Viewer.jar res/icon.pif
ou encore make run-view ARGS="res/icon.pif"

B. Le Convertisseur (Encodeur)

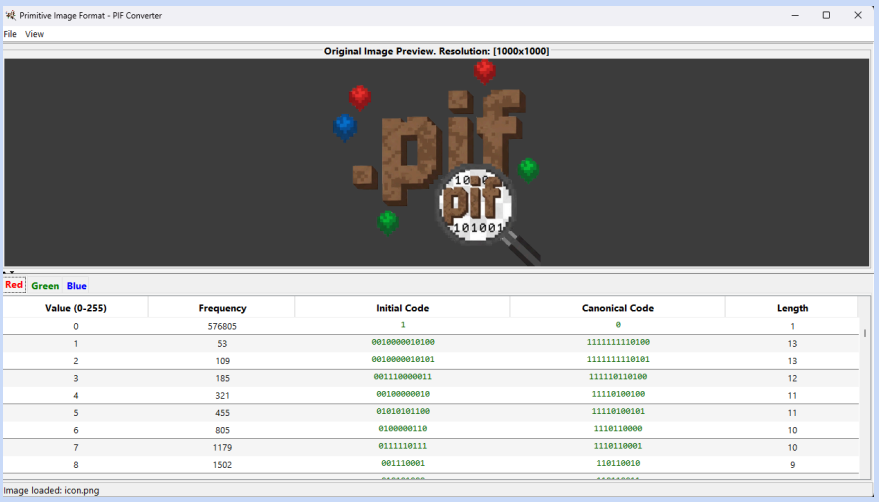
Une fois un fichier sélectionné (via le JFileChooser via le terminal) le convertisseur essaie de lire ce dernier : si le fichier n'est pas accepté, un message d'erreur s'affiche et l'utilisateur peut choisir un autre fichier via l'onglet "File" en haut à gauche.

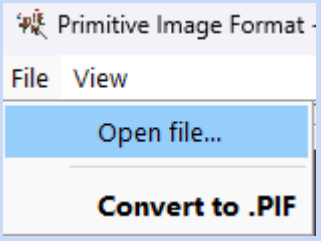


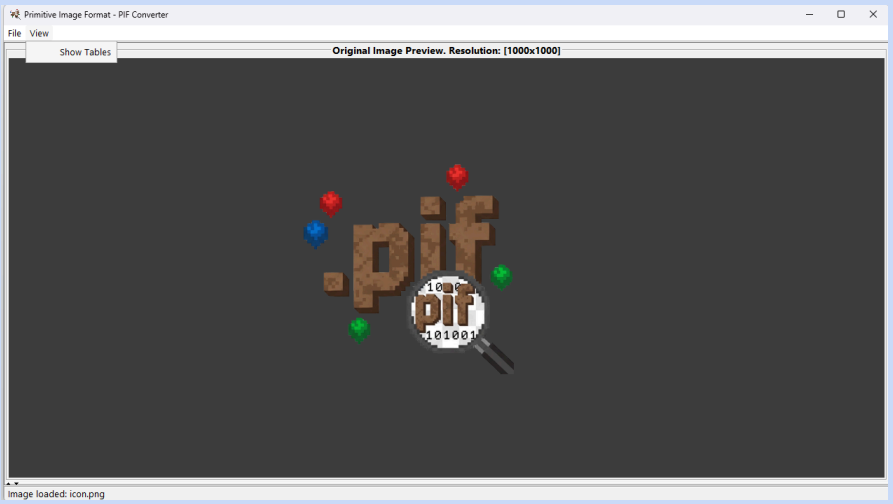




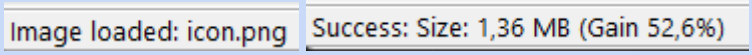
Lorsque l'image est acceptée, l'image est affichée sur le haut de la fenêtre et les tables de fréquences, codes initiaux ainsi que les codes canoniques sont affichés en bas. Ils sont triés par **valeur croissante** (de 0 à 255) et par **couleur**. Ceux de la couleur rouge sont sélectionnés par défaut mais l'utilisateur peut choisir d'afficher ceux du vert ou du bleu en cliquant sur la couleur souhaitée.







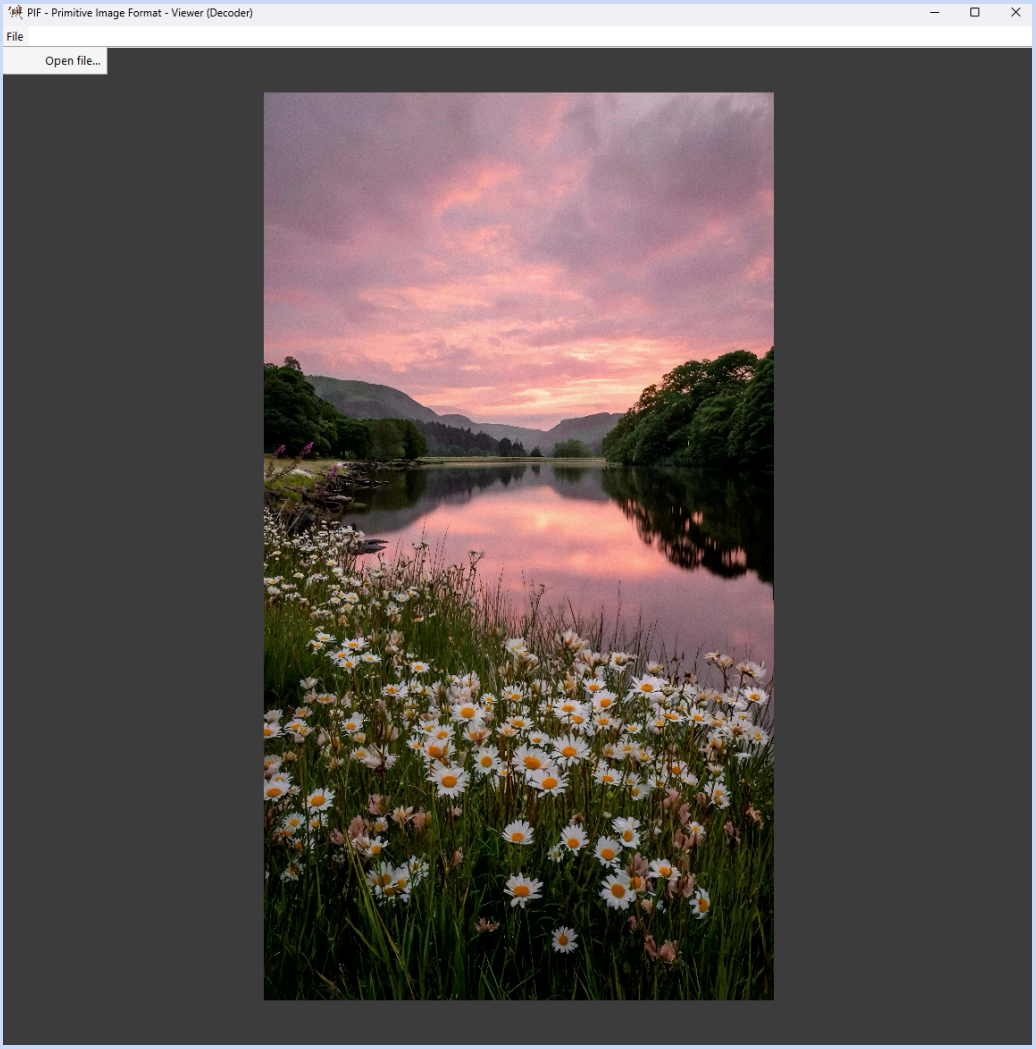
On a utilisé un **JMenuBar** afin d'avoir un minimum d'ergonomie et recharger une image depuis "File" ou encore pour convertir l'image actuelle en PIF.
Également, si l'utilisateur souhaite observer l'image un peu plus précisément, il peut décocher la case "*Show Tables*" dans "*View*".
Il peut ainsi **zoomer** sur l'image (le zoom s'effectue là où se trouve le curseur de la souris) et également se **déplacer** sur l'image avec les boutons de la souris.
On a également une fonctionnalité de Drag & Drop, où on peut glisser une image (valide) depuis le gestionnaire de fichiers dans la fenêtre.
Tout en bas, on retrouve une barre d'état :



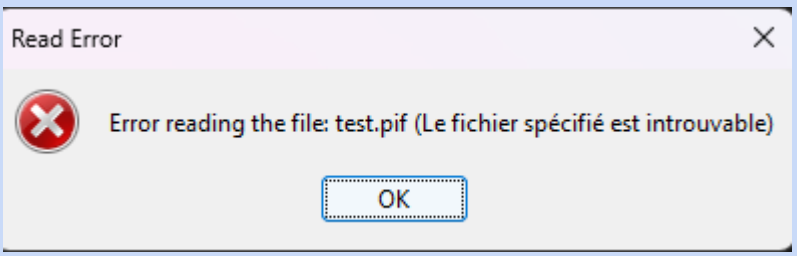
On peut ainsi observer l'état du convetisseur à chaque instant. C'est juste un choix ergonomique.

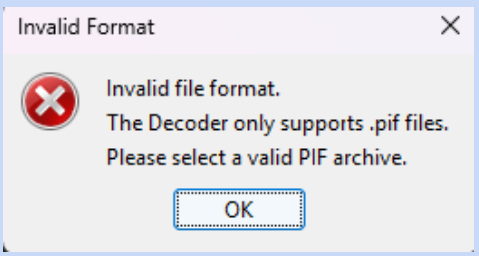
B. Le Visualisateur (Décodeur)

Le visualisateur reprend quasiment les mêmes mécaniques du convertisseur. Une fois un fichier .pif chargé depuis la ligne de commande (CLI) ou depuis l'interface JFileChooser (GUI), le fichier .pif est "décodé", l'arbre de Huffman est reconstitué et l'image est reformée. On peut également **zoomer** sur l'image et se **déplacer** dessus quand la **taille** de **l'image dépasse** la **taille** de la **fenêtre**.

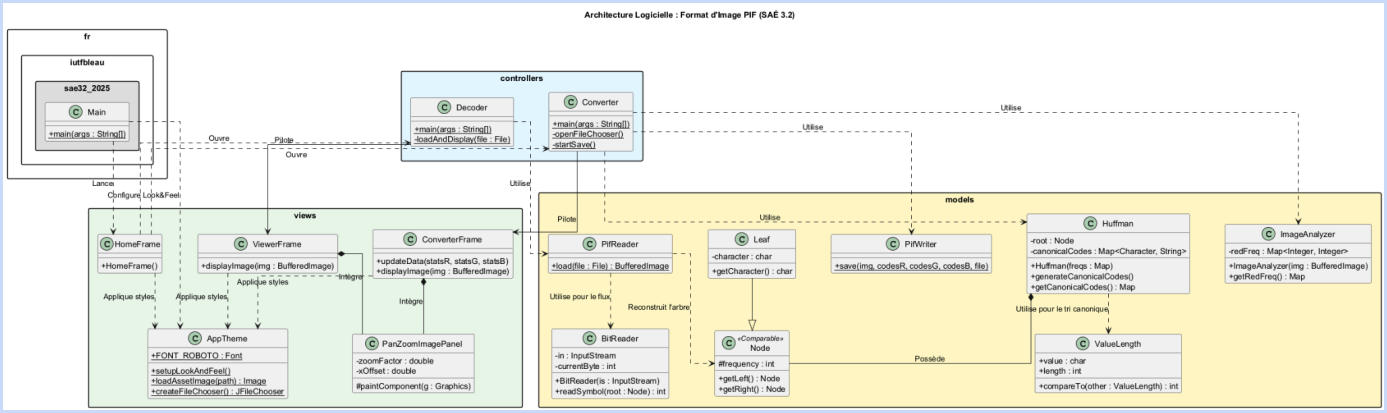


On communique encore une fois les erreurs avec l'utilisateur à l'aide d'un JPopUpPane : si le fichier PIF donné n'existe pas, si il est invalide…





III. Structure du code (Diagramme de Classes)



Le diagramme simplifié de nos classes à été réalisé à l'aide de plantuml et le résultat n'est pas vraiment adapté à une insertion dans le rapport. Par conséquent je vous conseille d'aller voir le fichier image du diagramme se trouvant dans **/diagrammes.** Ce diagramme comporte les méthodes et attributs principaux des classes que nous avons conçues.

Nous avons adopté une structure modulaire et rigoureuse avec le **MVC**. Le choix de l'anglais pour le nommage des classes, des variables et de la Javadoc a été privilégié car il s'agit de la norme internationale en informatique, garantissant que notre code est professionnel, éligible et compréhensible par d'autres développeurs.

L'architecture repose sur une séparation stricte des responsabilités, ce qui rend l'application maintenable, facile à déboguer et prête pour d'éventuelles extensions. Cette organisation s'articule autour de quatre piliers principaux :

- **Modèles (Models) :** Ils constituent le moteur du projet. Ils gèrent les structures de données (arbres binaires) et l'algorithmique pure (compression de Huffman). Ils sont totalement indépendants de l'interface et peuvent fonctionner sans elle.
- **Gestion de la Persistance (DAO) :** Bien que ce projet n'utilise pas de base de données relationnelle comme la précédente SAé, les classes `PifReader` et `PifWriter` agissent comme des **DAO** (Data Access Objects). Elles centralisent les opérations complexes de lecture et d'écriture binaire (Bit Packing), isolant ainsi le reste du programme des détails techniques du format de fichier.
- **Contrôleurs (Controllers) :** Ils servent de médiateurs. Ils réceptionnent les actions de l'utilisateur (clics, drag & drop, arguments de ligne de commande) et orchestrent le travail des modèles pour mettre à jour les vues en conséquence.
- **Vues (Views) :** Elles sont responsables uniquement de la présentation graphique. Elles utilisent les composants Swing pour offrir une interface ergonomique (zoom, panneaux divisés, tableaux statistiques) sans jamais intervenir dans les calculs de compression.
- **Charte Graphique et Utilitaires (AppTheme) :** Ce composant centralise les constantes visuelles (polices, couleurs, icônes) et la configuration du style natif du système, assurant une cohérence esthétique sur l'ensemble des modules.

1. La Couche Modèle (Data et Logique Métier)

Ces classes gèrent les structures de données (Arbres, Files de priorité), les algorithmes de compression (Huffman) et les opérations d'entrée/sortie binaires bas niveau.

Fichier	Rôle Détaillé (Concepts)	Application Concrète
ImageAnalyzer.java	**Analyste Statistique.** Scanne l'image pour établir la distribution des couleurs.	Parcourt chaque pixel, sépare les canaux RGB par opérations bit à bit (`>>`, `& 0xFF`) et remplit les `Map` de fréquences.
Huffman.java	**Moteur Algorithmique.** Implémente la logique de Huffman.	Construit l'arbre via une `PriorityQueue` (“Bottom-Up”), génère les codes binaires récursifs, et les transforme en **Codes Canoniques** pour optimiser le stockage.
PifWriter.java (DAO)	**Encodeur Binaire** Responsable de l'écriture du fichier `.pif` selon la spécification.	Écrit le Header (taille), les tables de longueurs, et effectue le **Bit Packing** : accumule les codes de longueur variable dans un buffer d'un octet avant écriture.
PifReader.java (DAO)	**Décodeur Binaire (Désérialisation).** Reconstruit l'image depuis le flux d'octets.	Lit les métadonnées, utilise les tables de longueurs pour reconstruire l'arbre canonique en mémoire, et décode les pixels via `BitReader`.

BitReader.java	**Utilitaire interne.** Abstraction de lecture bit à bit.	Java ne lit que des **octets** (8 bits). Cette classe lit un **octet**, le **stocke**, et **sert** les **bits** un par un au **décodeur**.
Node.java (POJO) & Leaf.java (POJO)	**Structures de Données.** Briques élémentaires de l'arbre binaire.	`Node` gère la structure (enfants gauche/droite). `Leaf` contient la valeur du pixel. Implémentent `Comparable` pour le tri par fréquence.
ValueLength.java (POJO)	**Tuple de Données.** Structure intermédiaire pour le format Canonique.	Associe une valeur (pixel) à la longueur de son code. Permet de trier les symboles pour reconstruire l'arbre sans ambiguïté.

Le défi principal résidait dans le décalage entre le fonctionnement de Java (qui écrit des fichiers par blocs d'octets de 8 bits) et la compression de **Huffman** (qui génère des codes de longueur variable, par exemple 3 ou 12 bits). Nous avons dû implémenter un système de "***Bit Packing***" (dans PifWriter) et de "***Bit Buffering***" (dans BitReader) pour aligner ces flux de données sans perdre d'information.
De plus, l'utilisation du **Huffman Canonique** a été pertinente.
Au lieu de sauvegarder la structure complexe de l'arbre binaire dans le fichier (ce qui prendrait trop de place), nous ne sauvegardons que la longueur des codes. Grâce à la classe ValueLength et aux règles de tri canonique, le décodeur est capable de reconstruire mathématiquement l'arbre exact utilisé à la compression, optimisant ainsi drastiquement la taille du fichier final.

2. La Couche Vue (Composants Graphiques Swing)

Ces classes sont responsables du rendu visuel, de l'ergonomie et de la capture des événements utilisateurs. Elles sont séparées de la logique métier complexe (compression/décompression) et servent l'affichage.
L'utilisation de JSplitPane n'était pas facile, mais ça devrait être bon.
La bibliothèque graphique utilisée est Swing.

Fichier	Rôle Détaillé (Composants/Ergonomie)	Application Concrète
HomeFrame.java	**Point d'entrée (Launcher).** Hub central de navigation. Utilise un design "flat" moderne avec des dégradés (`GradientPaint`). On a pas trouvé d'image de fond belle et pertinente.	Point d'entrée au lancement (`make run`). Permet de choisir entre le Convertisseur et le Visualisateur via des boutons stylisés (`createModernButton`).
ConverterFrame.java	**Interface d'Encodage Complexe.** Fenêtre divisée (`JSplitPane`) entre la prévisualisation et les données techniques. Intègre le Drag & Drop (`TransferHandler`).	Affiche l'image en haut et les tableaux de **Huffman** (Rouge/Vert/Bleu) en bas dans un `JTabbedPane`. Permet de masquer les tableaux pour passer en mode "Plein écran".
ViewerFrame.java	**Interface de Visualisation.** Fenêtre dédiée à l'affichage épuré. Gère le redimensionnement intelligent pour s'adapter	S'ouvre automatiquement à la taille de l'image décodée (avec des limites min/max). Contient le menu

	à la taille de l'écran de l'utilisateur.	"Fichier" pour ouvrir un nouveau PIF.
PanZoomImagePanel.java	**Composante d'interaction avec la souris**. Extension de `JPanel` gérant l'affichage matriciel via `AffineTransform` (Zoom centré souris, Panoramique).	Utilisé dans le `Viewer` (navigation libre) et le `Converter` (prévisualisation). Gère les événements souris (`MouseWheel`, `MouseDrag`) pour manipuler l'image en temps réel.
AppTheme.java	**Utilitaire de style**. Classe statique centralisant la charte graphique (Polices `Segoe UI`, couleurs…) et la configuration du "Look and Feel" système (on se débarrasse enfin de l'aspect “old school” que propose Swing de base…).	Appliqué au démarrage de chaque contrôleur pour garantir que l'application ressemble à un logiciel natif (Windows/Mac/Linux) et non à une vieille application Java.

3. La Couche Contrôleur (Logique de Flux et Événements)

Les contrôleurs font la médiation et implémentent la logique de l'application suite aux actions utilisateurs (en mode graphique ou console). Ils orchestrent les échanges entre les modèles (Huffman, Reader/Writer) et les vues.

Nous avons choisi de traiter le décodage et l'affichage de manière directe (synchrone). Bien que cela puisse immobiliser l'interface quelques millisecondes lors de fichiers très lourds, cette approche simplifie grandement la lecture du code et garantit que l'image ne s'affiche qu'une fois le décodage totalement terminé, évitant ainsi des erreurs d'affichage partiel. On a pensé à résoudre ce problème, mais cela impliquait l'utilisation de SwingUtilities.invokeLater, qui nous est interdit d'utiliser.

Fichier	**Rôle Détaillé (Logique)**	**Application Concrète des Événements**
Converter.java	**Chef d'orchestre de l'Encodage**. Centralise le cycle de vie de la compression : chargement de l'image, analyse statistique (`ImageAnalyzer`), création des arbres (`Huffman`) et écriture binaire. Gère aussi le mode CLI (depuis le terminal.	**Au lancement (CLI)** : `GuiInitializer` charge l'image et `AutoSaveAction` déclenche la sauvegarde automatique. **Via l'interface** : `OpenAction` ouvre le sélecteur de fichiers. `SaveAction` déclenche `PifWriter.save` pour générer le fichier binaire et met à jour la barre de statut.

Decoder.java	**Gestionnaire de Visualisation**. Point d'entrée dédié à la lecture sécurisée des fichiers PIF. Valide strictement l'extension du fichier et délègue le décodage complexe au modèle (`PifReader`).	**Au lancement** : Si un argument est présent, il appelle directement `loadAndDisplay`. Sinon, il force le filtre `.pif` dans le `JFileChooser`. **Après décodage** : Utilise `ViewerLauncher` (Runnable) pour instancier et afficher la `ViewerFrame` de manière Thread-Safe sur l'EDT.

4. La Classe Principale et Fichiers Annexes

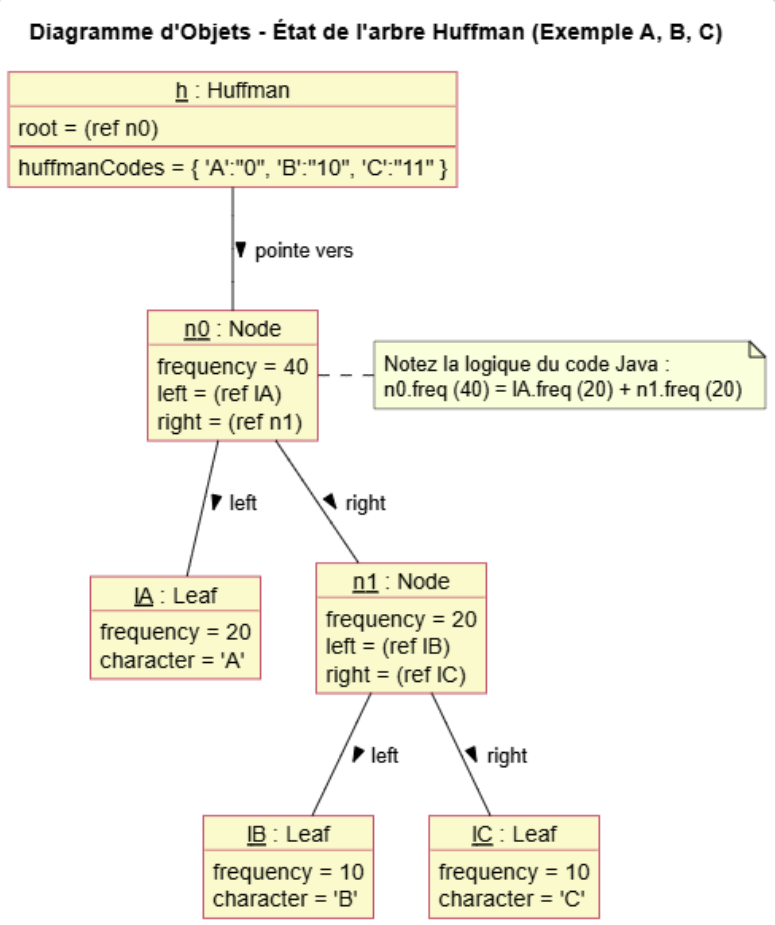
La gestion des ressources (images, icônes) une fois l'application compilée est cruciale. Si l'on charge une image avec un chemin standard comme `new File("res/icon.png")`, cela fonctionne pendant le développement, mais échoue systématiquement une fois le projet empaqueté dans un fichier JAR (car le fichier n'existe plus en tant que tel sur le disque, il est compressé dans l'archive).

Pour résoudre ce problème et garantir que nos JARs (`Converter.jar`, `Viewer.jar`) soient autonomes et exécutables sur n'importe quelle machine, nous avons adopté la méthode vue en cours :
Au niveau du Makefile : Nous copions explicitement le dossier `icons` dans le dossier de compilation (`build/`).
Au niveau du Code (`AppTheme`) : Nous n'utilisons pas de chemins de fichiers, mais le mécanisme de chargement via le *Classpath* (`AppTheme.class.getResource()`). Cela permet à Java d'aller chercher l'image à l'intérieur même du JAR, rendant les livrables totalement portables.

Fichier/Dossier	Description
Main.java	**Point d'Entrée Global**. Sa seule responsabilité est de configurer le "Look and Feel" système via `AppTheme`, puis d'instancier et d'afficher la fenêtre d'accueil (`HomeFrame`). Il agit comme un hub léger.
res/	**Ressources Graphiques**. Contient les icônes (`icon.png`) et l'image de fond non utilisée (`background.png`). Ces fichiers sont copiés dans le dossier `build` lors de la compilation pour être accessibles via le *Classpath*.
Makefile	**Script d'automatisation complet.** Il gère la compilation, l'exécution paramétrée (modes CLI/GUI), le nettoyage

	(`clean`) et la génération des trois archives JAR distinctes (`PIF_App`, `Converter`, `Viewer`).
diagrammes/	**Documentation Technique.** Contient les sources PlantUML (`.plantuml`) et les rendus visuels (`.png`) de l'architecture du projet (Diagramme de Classes).

IV. Exposition des classes composant l'arbre binaire & Diagramme d'objets



Ci-dessus, on peut voir que le code huffman de A est 0. Il est donc placé à gauche du nœud d'origine. B est en 10 donc il faut aller à droite puis à gauche dans l'arbre. Pour C c'est juste deux fois à droite.

L'arbre est composé de deux classes principales Node et Leaf.
Node (Le nœud structurel)

C'est la classe de base qui représente un point de connexion dans l'arbre.

- Rôle : Elle sert principalement de nœud interne (intermédiaire). Elle ne contient pas de valeur de pixel, mais relie d'autres nœuds.
- Données : Elle stocke la fréquence cumulée (frequency) de ses enfants et possède deux références : left (gauche) et right (droite).
- Comportement clé : Elle implémente l'interface Comparable. Cela permet de trier les nœuds du plus petit au plus grand selon leur fréquence, ce qui est indispensable pour construire l'arbre via la file de priorité.

Leaf (La feuille de données)

C'est une sous-classe de Node qui représente l'extrémité d'une branche.

- Rôle : C'est un nœud terminal. C'est le seul élément de l'arbre qui contient une information concrète à décoder.
- Données : En plus de la fréquence (héritée), elle ajoute un attribut character qui contient la valeur réelle du pixel (0-255).
- Logique : Quand on parcourt l'arbre avec des bits (0 ou 1), on s'arrête dès qu'on tombe sur une instance de Leaf.

Voici comment la classe Huffman agit sur ces deux types d'objets :

Elle les assemble. C'est elle qui instancie ces objets. Elle crée d'abord une Leaf pour chaque caractère présent dans l'image. Ensuite, elle les combine deux par deux en créant des Node parents (les nœuds internes) pour bâtir l'arbre complet, de bas en haut, jusqu'à la racine (root).

Elle les parcourt. Une fois l'arbre construit, Huffman traverse l'arbre récursivement pour générer les codes binaires :

- Si elle est sur un Node, elle continue de descendre (ajoute "0" pour gauche, "1" pour droite).
- Si elle arrive sur une Leaf (via instanceof ou isLeaf()), elle s'arrête et associe le code binaire construit au caractère de la feuille.

V. Structure des tables de codes & Mécanismes d'encodage et de décodage

1. Forme de la table des codes du convertisseur & Mécanisme d'encodage

A. Structure de la forme et de la table des codes

L'efficacité du format PIF repose sur la transformation des pixels (octets) en séquences binaires de longueurs variables. Cette transformation est rendue visible par la Table des codes du Convertisseur et opérée par le mécanisme de Bit-Packing.

Dans l'interface utilisateur (ConverterFrame), la table des codes n'est pas qu'un simple affichage de débogage ; elle représente la table qui sera utilisée pour la compression. Elle expose la relation directe entre la fréquence d'apparition des pixels et la longueur des codes générés. La table est structurée selon les colonnes suivantes :

1. Valeur (Symbole) : L'intensité du pixel (0 à 255). C'est la donnée brute à compresser.
2. Fréquence (Poids) : Le nombre d'occurrences du symbole dans le canal couleur analysé. C'est cette métrique qui détermine la position du nœud dans l'arbre de Huffman.
3. - Code Canonique : La séquence binaire finale attribuée au symbole.

La table met en évidence la propriété du Codage Canonique de Huffman. Contrairement à un arbre de Huffman standard (où le code

dépend de la structure topologique de l'arbre), les codes canoniques possèdent une structure mathématique stricte :

Les codes sont triés d'abord par longueur croissante (les symboles les plus fréquents ont les codes les plus courts).
À longueur égale, les codes sont triés par ordre lexicographique de leur valeur de symbole.

Cette forme permet de garantir que deux encodeurs produiront toujours strictement le même fichier binaire pour une même image, assurant le déterminisme de l'algorithme.

B. Mécanisme d'exploitation : Le "Bit-Packing"

L'exploitation de cette table pour l'écriture du fichier .pif pose un défi technique : le système de fichiers et les flux Java (OutputStream) fonctionnent par blocs de 8 bits (octets), alors que nos codes Huffman ont des longueurs arbitraires (ex: 101 fait 3 bits).

Nous avons implémenté un mécanisme de Bit-Packing (ou tamponnage de bits) au sein de la classe PifWriter. Ce mécanisme agit comme un tampon entre le flux binaire et le stockage physique réel.

Voici le processus étape par étape :

Mappage : Pour chaque pixel de l'image, le programme sépare les canaux RGB. Chaque composante (ex: Rouge = 128) est utilisée comme clé pour récupérer instantanément le code binaire associé dans la Map des codes canoniques (ex: 128 ⇒ "11010").

Accumulation dans le Buffer : Le code binaire récupéré est parcouru bit par bit. Chaque bit est injecté dans un tampon d'un octet (currentByte) via des opérations bit-à-bit :

- On décale le tampon à gauche : buffer = buffer << 1
- On insère le bit : buffer = buffer | bit

Vidage : Un compteur (bitCount) suit le remplissage du tampon. Dès que le compteur atteint 8 bits :

- L'octet complet est écrit physiquement sur le disque : dos.writeByte(buffer).
- Le tampon et le compteur sont remis à zéro.

À la fin du flux, si le tampon contient des bits “orphelins” (ex: 3 bits restants), ils sont décalés (padding avec des 0) pour former un octet complet avant la fermeture du flux, garantissant l'intégrité et le bon fonctionnement du fichier.

2. Forme de la table des codes du convertisseur & Mécanisme d’encodage

A. Structure de la forme et de la table des codes

Contrairement au convertisseur qui construit l'arbre à partir des fréquences observées dans l'image, le visualisateur (Viewer) ne dispose pas de l'image originale. Il reçoit un fichier binaire .pif et doit en déduire la structure de compression.

Dans le fichier et la mémoire du visualisateur, la table des codes ne stocke pas explicitement les séquences binaires (comme "11010") pour chaque valeur, car cela serait redondant et coûteux en espace. Grâce au **Huffman Canonique**, la table prend la forme compacte d'une **séquence de 256 octets** dans l'en-tête du fichier pour chaque canal (R, G, B). Chaque octet à l'indice *i* indique simplement la longueur du code pour la valeur de pixel *i*.

Données d'entrée : Une liste de longueurs (ex: Pour la valeur 0 ⇒ longueur 3, pour la valeur 1 ⇒ longueur 0, etc.).

Forme interne reconstituée : Le programme transforme cette liste brute en une structure triée (Longueur L, Valeur V) qui permet de régénérer les codes sans ambiguïté.

B. Mécanisme d’exploitation : Le Décodage et la Traversée d'Arbre

Le mécanisme exploite le déterminisme de l'algorithme canonique pour reconstruire l'arbre de décodage exact sans avoir besoin de connaître la topologie de l'arbre original. Ce processus inverse permet une symétrie avec l'encodage.

Le processus se déroule également en trois étapes séquentielles au sein de la classe `PifReader` :

1. **Reconstruction de l'Arbre (Mapping Inverse) :** Le lecteur lit les longueurs stockées dans l'en-tête. Il trie ensuite les valeurs selon les règles canoniques (d'abord par longueur croissante, puis par ordre lexicographique). En itérant sur cette liste triée, il assigne le premier code binaire disponible à la première valeur, puis incrémente le code pour la valeur suivante. Cela permet de régénérer l'arbre de Huffman exact qui a servi à la compression.
2. **Lecture du Flux (BitReader) :** Le corps du fichier `.pif` est une suite continue de bits. La classe `BitReader` inverse la logique du *Bit-Packing* : elle lit le fichier octet par octet, mais distribue les données bit par bit à l'algorithme de décodage, gérant le tampon de lecture de manière transparente.
3. **Traversée de l'Arbre :** Pour décoder un pixel, le mécanisme part de la racine de l'arbre reconstruit et suit le chemin dicté par le flux de bits :
    - Si le bit lu est `0`, le pointeur descend vers le nœud gauche.
    - Si le bit lu est `1`, le pointeur descend vers le nœud droit.
    - Dès que le pointeur atteint un objet de type `Leaf` (Feuille), la valeur du pixel (ex: 128) est récupérée et écrite dans l'image tampon (`BufferedImage`). Le pointeur est alors replacé à la racine pour décoder le pixel suivant.

Ce mécanisme garantit que l'image décodée est strictement identique à l'originale (compression de type *lossless*), validant ainsi l'intégrité de la chaîne de traitement PIF.

PS : Il est plus intéressant de tester le programme de compression (Convertisseur) sur un fichier image sans perte du style .bmp. Les fichiers .jpg et .png sont compressés avec perte (donc très légers) et

parfois, les fichiers .pif peuvent être plus lourds qu'une image .png de base (qui "bénéficie" de la compression avec perte).

VI. Conclusion Personnelle

Luka PLOUVIER:

Comparé aux autres SAÉs, j'ai vraiment été inspiré par le sujet. J'avais vraiment à cœur de prendre des initiatives dans cette SAé et j'ai vraiment été surpris que mes camarades soit satisfait du travail que j'ai proposé, même si tout n'était pas parfait, j'ai apprécié voir les corrections qu'ils m'ont apportées. De plus, je sais que je suis moins à l'aise en codage que mes camarades alors ça m'a encore plus motivé pour en faire d'avantages.
Cette SAé été tout de même très compliquée à comprendre et à commencé, mais une fois lancé nous avons vraiment eu des facilité à communiquer, cette communication était très fluide ce qui nous à permis de bien avancé dans cette SAÉ sans avoir d'interruption. Développer deux programmes peut sembler être impossible mais avoir pris chaque programme et les avoir découpés nous à vraiment soulager et nous avons pu nous concentrer petit à petit pour arriver à cette finalité qui pour ma part me semble le meilleur que nous puissions faire.
C'est vraiment satisfaisant de voir l'évolution pas à pas de notre projet.
Pour finir, travailler avec Canpolat et Maxime m'a permis de vraiment me sentir aidé et écouté, je suis très content d'avoir fait cette SAÉ à leur côtés ils m'ont beaucoup expliqués et beaucoup appris sur des notions que je ne maîtrisait pas forcément.

Canpolat DEMIRCI–ÖZMEN :

Au début, le sujet de cette SAé était assez effrayant. Ce projet remontait jusqu'aux flux d'octets qu'on avait vu au S2, donc un rappel s'imposait. De plus, aborder des concepts qu'on a pas vu en cours comme le codage de Huffman ou encore les codes canoniques, c'était un challenge. Heureusement, on a eu le droit à des explications de la part de Florent Madelaine (grand merci à lui !) concernant Huffman lors

de son cours de DEV3.4 qui nous a rassuré sur la faisabilité de ce projet. Cette SAé était la plus “technique” qu'on ait pu réaliser (à mon avis) : ce n’était pas un jeu, ni une application basée sur l’UX.
Mais un convertisseur, similaire à ce qu’on peut trouver sur le web et qu’on utilise tous les jours (jpg to png, webp to png…) sur un format assez particulier : le PIF. Plus on s’approchait du code, plus le projet et la façon dont on pouvait réaliser l’encodeur comme le décodeur nous paraissait. La synergie au sein de notre groupe facilitait notre avancée et notre compréhension, la communication était fluide, la confiance était très bonne puisque chacun honorait ses responsabilités dans ce projet. Je suis content que Luka se soit senti à l’aise, il était très pertinent tout au long du projet. De même, c’est un plaisir de retravailler avec Maxime, qui nous a bien éclairci sur le code canonique. L’enchaînement de la semaine de partiels, des fêtes et des vacances (où Luka et moi étions malades) nous a un peu ralenti. Mais malgré ça, c'est durant cette SAé que j’ai le + remarqué l’importance de travailler en groupe avec des personnes dynamiques et pertinentes autour d’un projet en commun.

Maxime ELIOT :

Encore une fois, c’est un plaisir de pouvoir développer une application différente d’un jeu. Je suis content du résultat que nous avons produit.
Le fait d’écrire bit à bit dans le fichier pour la conversion nous a pris du temps de conception plus conséquent que ce qu’on pensait mais cela ne nous a pas empêché de finir en temps et en heure ! L’algorithme d'optimisation était intéressant à concevoir surtout au niveau du passage des codes Huffman vers les codes canoniques puisque nous avons pu revoir l’utilisation de la classe Comparator que je n’avais pas bien assimilée auparavant. Les travaux de groupes sont toujours un bon moyen de comprendre mieux les notions devant être acquises et pour ma part je trouve que ce fut une réussite aussi de ce côté.