

INSA de Rennes
Département INFORMATIQUE

Rapport final

Encadrants : Marie BABEL, Ivan LEPLUMEY

Projet DETIQ-T

Benoit AVERTY, Samuel BABIN
Matthieu BERGÈRE, Thomas LETAN
Sacha PERCOT-TÉTU, Florian TEYSSIER

Rennes, le 30 mai 2012

Table des matières

Introduction	iii
1 Spécifications	1
1.1 Bibliothèque IMAGEIN	1
1.2 Interface générique	2
1.3 Applications pédagogiques	2
1.3.1 Application de masques binaires	2
1.3.2 Filtrage	2
1.3.3 Binarisation	3
1.3.4 Étiquetage des composantes connexes	3
1.3.5 Morphologie mathématique	4
1.3.6 Projection	5
2 Conception	7
2.1 Bibliothèque IMAGEIN	7
2.2 Interface générique	9
2.3 Applications pédagogiques	15
2.3.1 Application de masques binaires	15
2.3.2 Filtrage	15
2.3.3 Binarisation	15
2.3.4 Étiquetage des composantes connexes	15
2.3.5 Morphologie mathématique	16
3 Phases de tests	17
3.1 Tests de IMAGEIN	17
3.1.1 Tests manuels	17
3.1.2 Tests automatiques	18
3.2 Tests de l'interface générique	21
3.3 Tests des applications	22
4 Finalisation du projet	25
4.1 Bibliothèque IMAGEIN	25
4.2 Interface générique	26
4.3 Applications pédagogiques	27

4.3.1	Application de masques binaires	27
4.3.2	Filtrage	27
4.3.3	Binarisation	27
4.3.4	Étiquetage des composantes connexes	28
4.3.5	Morphologie mathématique	28
Conclusion		28
A Manuel d'utilisation		31
A.1	Bibliothèque IMAGEIN	31
A.1.1	Utilisation de la bibliothèque	31
A.1.2	Fonctionnalités avancées	34
A.1.3	Étendre la bibliothèque	35
A.2	Interface générique	38
A.2.1	Comportement pré-défini	38
A.2.2	Utilisation dans un projet	40
A.2.3	Créer un widget DETIQ-T	42
A.3	Compilation	42
A.3.1	Pré-requis	43
A.3.2	Compilation de IMAGEIN	43
A.3.3	Compilation de l'interface générique	43
A.3.4	Compiler l'application unitaire	45

Introduction

La première version de DETIQ-T est enfin fonctionnelle et disponible. Après un an de dur labeur, nous allons faire le point entre les prévisions initiales et l'aboutissement final.

DETIQ-T est un projet d'un an mené par un groupe d'étudiants de quatrième année du département Informatique de l'INSA de Rennes, encadrés par Ivan LEPLUMEY, directeur, chercheur et enseignant au département Informatique, et Marie BABEL, chercheuse et enseignante au département EII (Électronique et Informatique Industrielle).

Les objectifs initiaux étaient la création d'une bibliothèque extensible d'algorithmes de traitement d'images, la réalisation d'une interface générique, également extensible, l'implémentation d'applications pédagogiques basées sur la bibliothèque et l'interface précédente et la mise en place d'une application plus conséquente, nommée MERLIN, pour valider la structure du projet. Ces objectifs ont finalement été revus au cours de l'année. Nous avons en effet été trop optimistes et nous avons dû renoncer à MERLIN. Les raisons de ce choix sont principalement expliquées dans le rapport de retour sur la planification.

Le présent rapport a pour objectif de permettre au lecteur de pouvoir appréhender, utiliser et reprendre notre projet. Nous avons fait de notre mieux pour qu'il se suffise à lui-même, afin d'épargner au potentiel contributeur d'autres lectures. Dans un premier temps, nous verrons les modifications que nous avons apportées aux spécifications que nous avons imaginées au mois de novembre de l'année dernière. Certaines ont été enlevées puisqu'elles ne nous paraissaient plus très pertinentes, ou encore par manque de temps, alors que d'autres ont été ajoutées pour améliorer l'ergonomie des applications. Ensuite, nous verrons les évolutions qu'a subies notre conception initiale et les raisons qui nous ont poussés à la modifier. Les explications de nos phases de tests suivront, et nous finirons par présenter l'état final du projet, en expliquant où nous en sommes arrivés et ce qu'il reste à faire. Nous y proposerons quelques améliorations pouvant être effectuées sur l'ensemble du projet. Étant complètement extensible, il est impossible de donner toutes les évolutions possibles.

En annexe de ce document se trouve un manuel utilisateur expliquant non seulement le fonctionnement de nos applications, mais aussi l'utilisation, d'un point de vue développeur, de notre bibliothèque et de notre interface générique, pour permettre à tous d'implémenter ses algorithmes et ses propres applications.

Chapitre 1

Spécifications

Dans un précédent rapport, nous avons détaillé les spécifications de DETIQT. Maintenant que le développement des applications pédagogiques est terminé, nous avons assez de recul pour critiquer, et au besoin ajuster, nos volontés initiales.

1.1 Bibliothèque IMAGEIN

Objectifs principaux

Nous avons comme objectif de développer une bibliothèque portable, extensible et open-source. A l'exception des classes de lecture et écriture de fichiers, notre bibliothèque n'utilise que du code C++ standard, elle est donc portable sur n'importe quelle plateforme supportant un compilateur C++. Seule l'utilisation des formats JPEG ou PNG nécessitent de disposer de bibliothèques standards (respectivement libjpeg et libpng), le format BMP pouvant fonctionner sans bibliothèque externe. Notre bibliothèque est également très axée sur la généricité, toutes les structures de données, ainsi que les classes représentant les images et les algorithmes sont génériques et peuvent donc être utilisées pour d'autres types d'images et d'algorithmes que ceux initialement prévus. **ImageIn** doit donc être une bibliothèque très facilement extensible et peut être adaptée pour s'utiliser avec presque tous les types d'images matricielles et d'algorithmes possibles. Enfin, notre bibliothèque n'intégrant aucun code source propriétaire, nous avons la possibilité de la distribuer sous licence open-source.

Objectifs secondaires

Nos trois spécifications principales ont donc bien été respectées. Seules quelques modifications mineures ont été apportées aux autres spécifications initiales, notamment la suppression du format WebP jugé encore trop jeune et trop peu répandu. Il sera cependant très simple de l'ajouter par la suite, de par l'extensibilité très poussée de la bibliothèque, à tous les niveaux. La section A.1.3 du présent rapport présente d'ailleurs en détails la marche à suivre pour parvenir

à ce résultat. Nous avons également parlé dans les spécifications de différents espaces colorimétriques. Nos applications n'utilisent finalement que les espaces RGB et niveaux de gris, à 8 bits par canal, mais il est tout à fait possible d'imaginer l'utilisation d'autres systèmes colorimétriques, de par la généricité de la bibliothèque. Il suffirait alors simplement d'implémenter les formats de fichiers correspondants.

1.2 Interface générique

Les spécifications générales de l'interface générique ont été respectées. Nous retrouvons bien notre barre de menu en haut de la fenêtre principale, et le système de la barre de navigation, contenant les miniatures des images ouvertes, à gauche. L'utilisation d'une zone MDI (Multiple Document Interface) nous permet bien d'ouvrir plusieurs fenêtres (images, histogrammes, grilles de pixels, etc...) au sein même de l'application.

Certaines fonctionnalités ont même été ajoutées par rapport à nos prévisions. En effet, l'affichage des histogrammes et de la grille de pixels est maintenant géré directement par l'interface générique. L'ensemble de ces fonctionnalités est donc disponible à toutes les applications qui seront créées avec cette interface.

1.3 Applications pédagogiques

1.3.1 Application de masques binaires

L'application **BitPlane** remplit tout à fait les spécifications que nous nous étions fixées. Il était prévu d'offrir deux possibilités d'affichage : soit générer une image séparée de l'image originale, soit lui superposer le résultat. Ici, c'est la première solution qui a été implémentée, on peut donc facilement comparer le résultat par rapport à l'image d'origine. La Figure 1.1 présente une capture d'écran partielle de cette application avec à gauche l'image d'origine, et à droite les 2 fenêtres permettant d'appliquer un masque binaire.

1.3.2 Filtrage

- L'application de filtrage, nommée *FiltrMe* avait deux principaux objectifs :
- appliquer des filtres sur une image ;
 - créer ses propres filtres (en plus de ceux par défaut).

Les filtres les plus connus, tels que *Sobel*, *Roberts*, *Prewitt* ou encore *Gaussien*, ont été implémentés et sont donc disponibles par défaut dans cette application. De plus, un système de création et d'enregistrement de filtres personnels a également été créé. Il permet à l'utilisateur de rentrer ses propres filtres, de les sauvegarder et de les appliquer facilement.

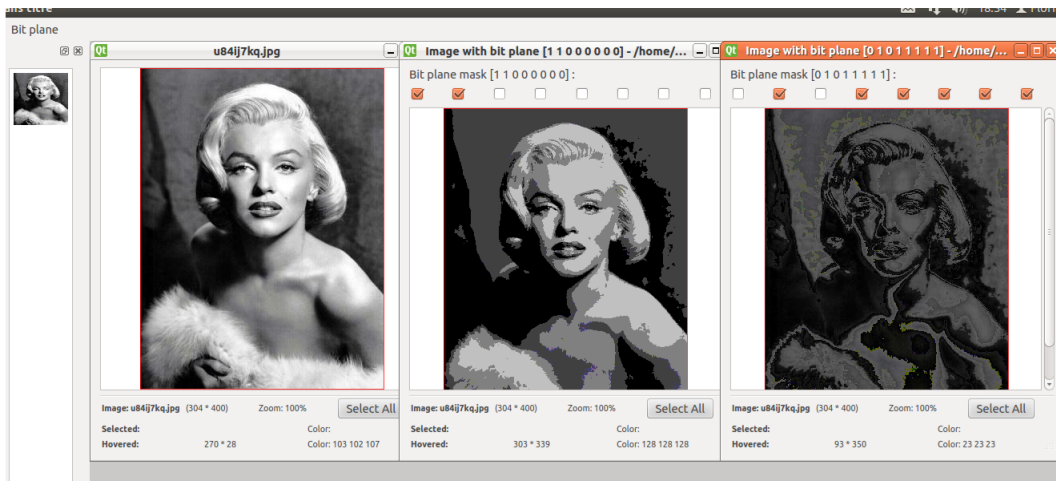


FIGURE 1.1 – Capture d’écran partielle de l’application BitPlane

Une dernière petite fonctionnalité avait été évoquée, mais n’a pas été réalisée par manque de temps. Il s’agit de la visualisation de la fonction de transfert des filtres.

1.3.3 Binarisation

L’application de binarisation répond aux spécifications, à savoir appliquer un algorithme de binarisation ainsi que donner le seuil calculé. Comme décidé pendant la phase de conception, on peut appliquer la méthode Otsu ou bien choisir son ou ses propres seuils.

Il aurait pu être intéressant de développer un autre algorithme, qui aurait été intégré facilement à l’application. En outre, dans les spécifications il était question de faire une binarisation à la volée sur l’histogramme, ce qui n’est pas encore réalisé. Néanmoins la binarisation se fait en direct et on peut voir l’histogramme juste en dessous du choix du seuil, ce qui est proche de la spécification initiale.

1.3.4 Étiquetage des composantes connexes

L’objectif principal de cette application était de pouvoir appliquer l’algorithme d’étiquetage des composantes connexes. Cet objectif a bien été réalisé puisqu’il est possible d’appliquer l’algorithme en réglant les deux paramètres (connexité, couleur du fond/des objets).

En revanche, il n’est pas possible d’appliquer l’algorithme directement, comme il était proposé dans les spécifications initiales. En effet, il est impératif de régler

les paramètres avant l'application de l'algorithme (ou au moins de les visualiser). L'application a été développée de cette manière car il est important de connaître les paramètres utilisés pour appréhender le résultat.

Les données secondaires d'application de l'algorithme (nombre de composantes identifiées, taille des composantes) sont aussi moins nombreuses que prévues. Des données supplémentaires sont toujours en projet pour cette application, par exemple la distance moyenne entre les composantes ou d'autres mesures.

Il n'est pas non plus possible de n'appliquer que la première passe de l'algorithme puisque l'algorithme présent dans la bibliothèque `ImageIn` ne gère pas cette fonctionnalité. Encore une fois, il s'agit d'une fonctionnalité qui a été mise de côté et non pas abandonnée.

1.3.5 Morphologie mathématique

L'application `MorphoMat` avait pour objectif de permettre à l'utilisateur de se familiariser avec les principes de la morphologie mathématique.

L'application ne devait initialement fonctionner qu'avec des images en niveaux de gris. Cependant, les algorithmes de morphologie ayant été implémentés pour s'appliquer canal par canal, l'application finale traite les images de manière générique indépendamment du nombre de canaux. Il est donc possible d'effectuer des opérations de morphologie mathématique sur des images couleurs par exemple.

L'utilisateur devait avoir une certaine liberté quant au choix de l'élément structurant. La possibilité de choisir l'élément parmi les éléments prédéfinis a bien été implémentée, ainsi que la fonctionnalité permettant d'ouvrir et d'enregistrer l'élément structurant à partir et vers un fichier image. Deux fonctionnalités supplémentaires ont été mises en places :

- redimensionner l'élément structurant ;
- choisir le centre de l'élément structurant (décentrer).

Les opérateurs d'érosion et de dilatation ont bien évidemment été implémentés, ainsi qu'un certain nombre d'autres opérateurs morphologiques tels que l'ouverture, la fermeture, le gradient (détecteur de contour), le top-hat blanc (extraction des éléments clairs) et le top-hat noir (extraction des éléments sombres). Cependant la mise en place de nouveaux algorithmes par combinaisons d'opérateurs existants et l'application étape par étape n'ont pas été implémentées. Cela aurait nécessité le développement d'un certain nombre de fonctionnalités qui ne sont pas incluses dans l'interface générique. Il reste cependant possible d'appliquer successivement plusieurs opérateurs sur une même image et ainsi visualiser progressivement le résultat, ce qui n'est cependant pas équivalent.

1.3.6 Projection

L'application de projection n'a finalement pas été implémentée. Les histogrammes de projection font partie intégrante de l'interface générique et peuvent donc être affichés depuis n'importe quelle application. La méthode de seuillage que nous proposons dans l'application de projection ne semblait dès lors que peu intéressante comme unique plus-value.

Si certains objectifs ont été revus à la baisse, nous pouvons tout de même remarquer que nos principaux axes de spécification restent inchangés.

Chapitre 2

Conception

La conception que nous avons détaillée dans notre précédent rapport ayant été longuement réfléchie, peu de points ont nécessité une correction lors de l'implémentation. Le détail de ces corrections est présenté ci-dessous.

2.1 Bibliothèque IMAGEIN

La bibliothèque `ImageIn` est restée sensiblement conforme à celle décrite dans le rapport de conception.

BinaryImage :

La classe `BinaryImage` qui héritait de `GrayscaleImage_t` a finalement été retirée car elle apportait plus de contraintes que de solutions. En effet notre système d'algorithmes impose que la profondeur des images en entrée soit la même que celle en sortie, ce qui était problématique notamment pour la binarisation.

Désormais les images binaires seront représentées par des `GrayscaleImage_t` (comme c'était le cas dans notre conception : une `GrayscaleImage_t` avec profondeur booléenne), mais la liberté est maintenant laissée au développeur de choisir la profondeur de l'image : il est toujours possible de créer une `BinaryImage` en choisissant une profondeur booléenne, mais on ne force plus l'utilisateur à utiliser cette profondeur.

Ajout de typedefs :

Plusieurs typedefs ont été ajoutés pour simplifier la manipulation des profondeurs et des classes templâtées associées.

- `depth8_t` pour `uint8_t`
- `depth16_t` pour `uint16_t`
- `depth32_t` pour `uint32_t`

- *depth_default_t* pour *depth8_t*
- *GrayscaleImage_8* pour *GrayscaleImage_t<depth8_t>*
- *GrayscaleImage_16* pour *GrayscaleImage_t<depth16_t>*
- *GrayscaleImage_32* pour *GrayscaleImage_t<depth32_t>*
- *GrayscaleImage* pour *GrayscaleImage_t<depth_default_t>*
- *Image_8* pour *Image_t<depth8_t>*
- *Image_16* pour *Image_t<depth16_t>*
- *Image_32* pour *Image_t<depth32_t>*
- *Image* pour *Image_t<depth_default_t>*
- *ProjectionHistogram_8* pour *ProjectionHistogram_t<depth8_t>*
- *ProjectionHistogram_16* pour *ProjectionHistogram_t<depth16_t>*
- *ProjectionHistogram_32* pour *ProjectionHistogram_t<depth32_t>*
- *ProjectionHistogram* pour *ProjectionHistogram_t<depth_default_t>*
- *RgbImage_8* pour *RgbImage_t<depth8_t>*
- *RgbImage_16* pour *RgbImage_t<depth16_t>*
- *RgbImage_32* pour *RgbImage_t<depth32_t>*
- *RgbImage* pour *RgbImage_t<depth_default_t>*

Implémentation des graphes :

Certains algorithmes nécessitent de manipuler des graphes, notamment des graphes représentant les liens entre pixels (voir Figure 2.1), par exemple en utilisant des graph-cuts¹. C’est pourquoi une classe **Graph** a été ajoutée à **ImageIn**.

Cette classe simpliste permet de représenter des graphes valués et/ou dirigés mais ne fournit par pour le moment d’outils avancés de gestion des graphes (comme par exemple les graph-cuts mentionnés ci-avant). Elle sert surtout de base pour une éventuelle utilisation future pour des algorithmes tels que la texturisation qui se sert des graphes pour déterminer comment sélectionner les pixels à recopier pour propager un motif.

La classe est construite à partir du nombre de ses sommets et fournit les méthodes les plus simples de manipulation des graphes (ajout et suppression d’arêtes, vérification de l’existence d’une arête, et vérification de la capacité d’une arête).

Convertisseur :

Une classe **Converter** a été ajoutée à **ImageIn** pour permettre à l’utilisateur de convertir facilement un type d’image en un autre. Cette classe est utile lorsqu’on dispose par exemple d’une **GrayscaleImage_t** et que l’on a besoin d’une **RgbImage_t**, ou bien si l’on souhaite généraliser l’image en la convertissant en **Image_t**.

1. Graph-cut est un algorithme qui fournit une solution optimale pour le problème de la coupe maximale d’un graph

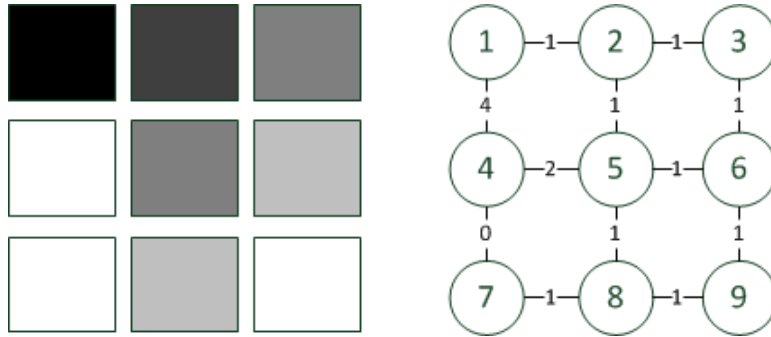


FIGURE 2.1 – Exemple de représentation sous forme de graphe d’une image de 3x3px avec 5 couleurs, les valuations des arêtes représentent le gradient entre les couleurs des pixels adjacents

Système d’algorithmes :

La couche algorithmes de **ImageIn** a subi une refonte importante. Cette refonte permet un meilleur contrôle de l’arité des algorithmes. Le nouveau système est celui décrit par la Figure 2.2.

Les modifications effectuées n’entament que peu la méthode d’utilisation décrite dans les précédents rapports. Les classes avec lesquelles interviendra l’utilisateur sont essentiellement **AlgorithmCollection_t** et **Algorithm_t**. Le template **D** correspond à la profondeur des images, le template **I** correspond au type d’image de sortie et le template **A** représente le nombre d’images en paramètres de l’algorithme.

Pour utiliser **AlgorithmCollection_t**, on donne au constructeur un algorithme de profondeur **D** et d’arité **A**, et les algorithmes qui constituent la chaîne devront tous avoir cette même profondeur **D** mais avoir une arité de 1. En effet le premier algorithme prendra **A** images et rendra une image résultat qui passera ensuite au travers des autres algorithmes de la collection. **Algorithm_t** reste la classe de laquelle hériteront les algorithmes implémentés par l’utilisateur.

2.2 Interface générique

Tout comme pour **ImageIn**, nous n’avons pas eu beaucoup de modifications à apporter à la conception de notre interface générique. Le système de Services a été respecté et tous ceux disponibles par défaut ont été implémentés sur le modèle imaginé lors de la conception. Les seules modifications que nous avons apporté sont mineures et sont résumées ci-après.

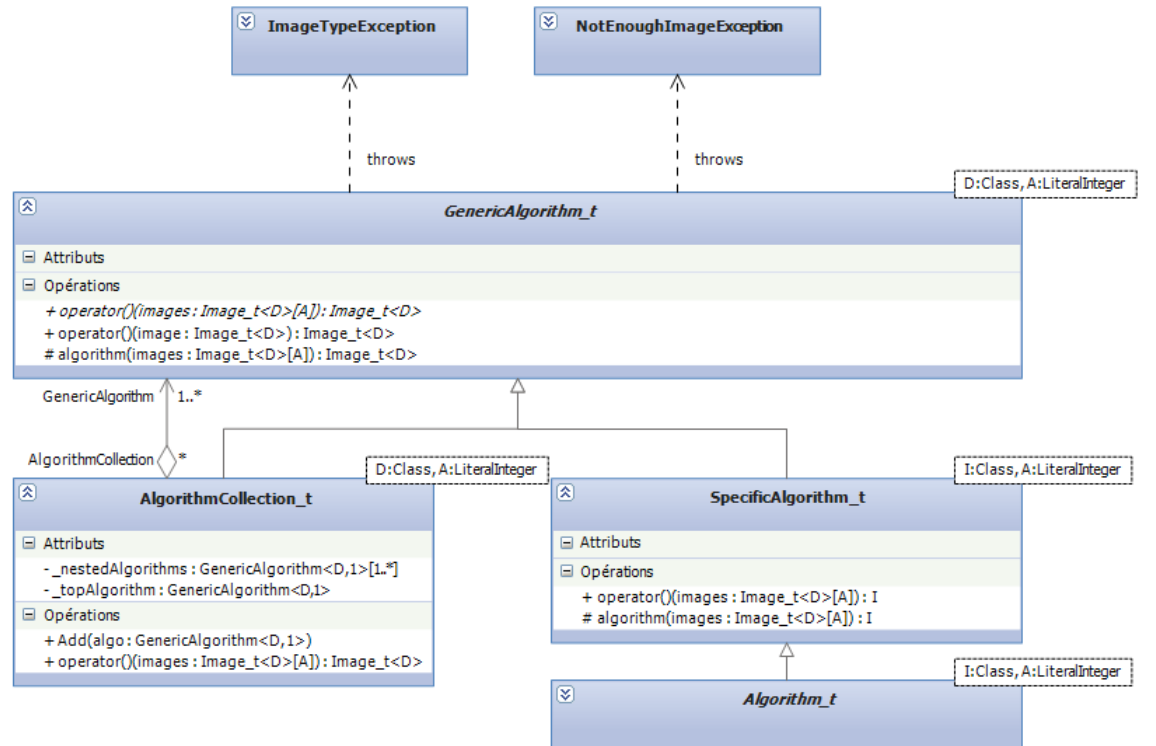


FIGURE 2.2 – Diagramme de classes du système d'algorithmes

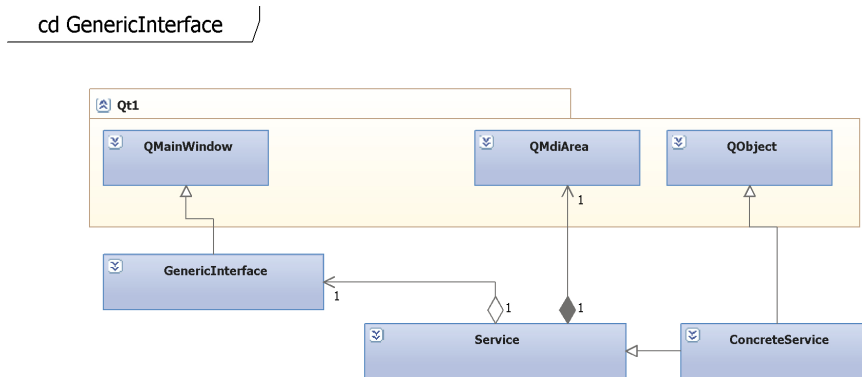


FIGURE 2.3 – Diagramme de classes allégé de l'interface générique

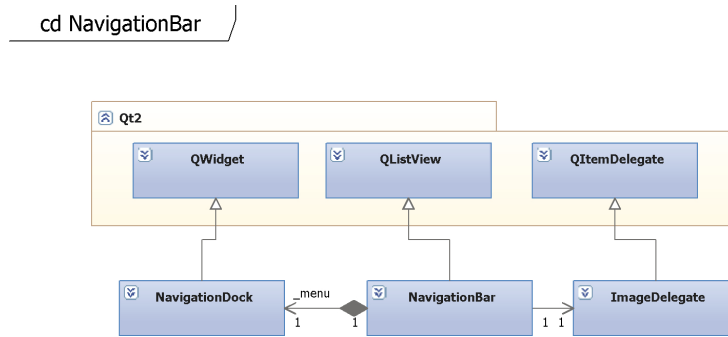


FIGURE 2.4 – Diagramme de classes allégé de la barre de navigation

MainWindow :

La classe que nous avons appelée **MainWindow** a été remplacée par une classe **GenericInterface**, mais son but et son comportement sont restés les mêmes. De plus, c'est maintenant le **WindowService** qui gère la zone centrale en tant que **QMdiArea** simple (la classe **CentralZone** a été supprimée). Le nouveau diagramme de classes est représenté par la Figure 2.3.

Barre de navigation :

La classe **MiniatureDelegate** a juste été renommée en **ImageDelegate**, et **QNavigationBarMenu** par **NavigationDock**. Cette dernière hérite maintenant de **QWidget** et plus de **QMenu**, et permet d'ouvrir ou de fermer des images et de récupérer la liste des fenêtres ouvertes dans la zone centrale. La Figure 2.4 représente le diagramme de classes corrigé.

Affichage des histogrammes :

C'est ici qu'il y a eu le plus de modifications apportées par rapport à la conception initiale. Premièrement, les histogrammes de projection n'étaient pas présents dans notre première version de conception. Deux classes ont été ajoutées : `ProjectionHistogramWindow` et `View`. Ensuite nous nous sommes aperçu que les comportements des classes `HistogramView`, `RowView` et `ProjectionHistogramView` étaient quasiment identiques. Nous avons donc décidé de toutes les faire hériter d'une nouvelle classe, appelée `GenericHistogramView`.

Enfin, le choix d'utiliser Qwt² pour l'affichage des histogrammes et autres graphes nous a poussé à ajouter une classe `GraphicalHistogram` héritant de `QwtPlotHistogram`. Une autre a également dû être créée pour gérer les clics de souris sur les histogrammes. Il s'agit de la classe `HistogramPicker` qui hérite de `QwtPlotPicker`.

Grille de pixels :

La grille de pixel s'est révélée être plus compliquée à gérer que prévu, il nous a fallu ajouter deux classes pour l'affichage et la gestion de la souris sur l'image miniature sur le bord de la grille. Il s'agit de `ImageViewer` et de `ZoomViewer`.

La nouvelle version du diagramme de classes est représentée par les Figures 2.5 et 2.6.

2. Qwt (ou Qt Widgets for Technical Applications) est un ensemble de widgets Qt

cd ImageView

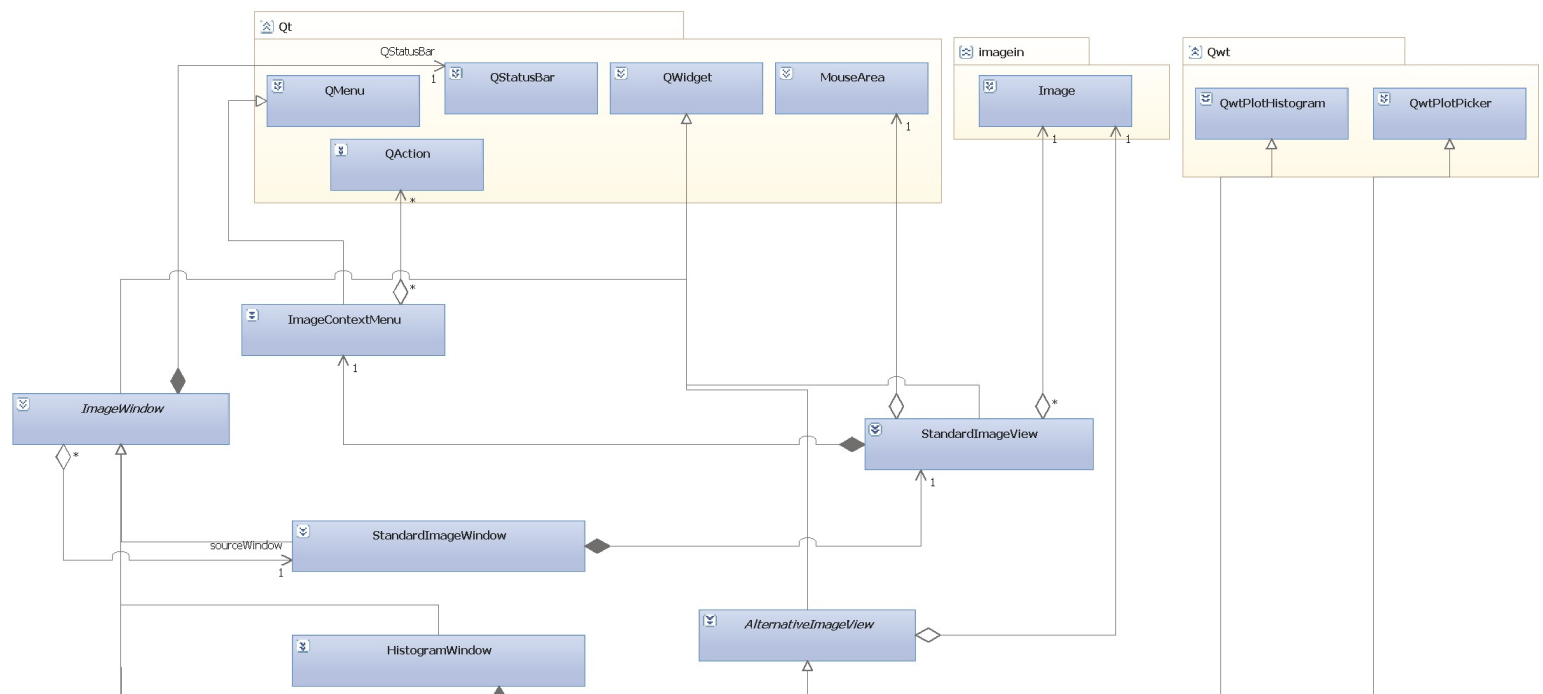


FIGURE 2.5 – Diagramme de classes allégé de l’affichage des images (1/2)

2.2.

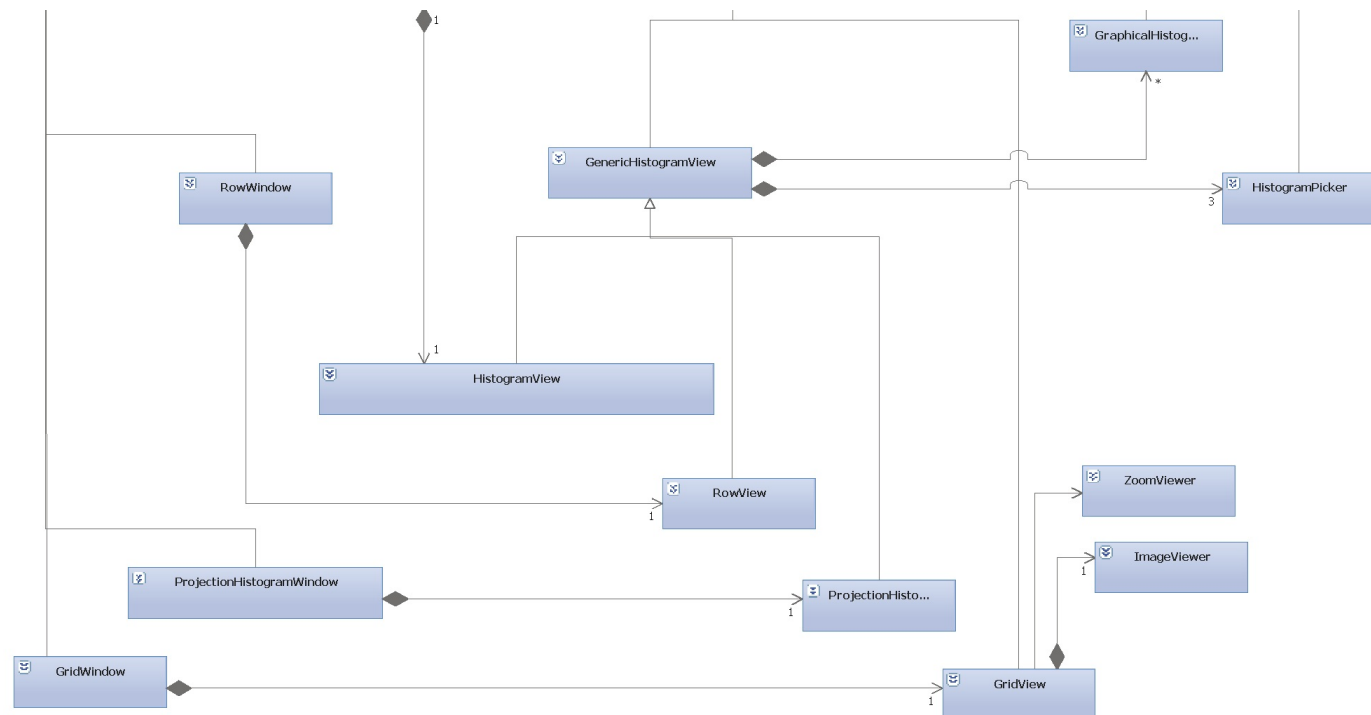


FIGURE 2.6 – Diagramme de classes allégé de l’affichage des images (2/2)

2.3 Applications pédagogiques

2.3.1 Application de masques binaires

Cette application, nommée `BitPlane` (et non plus `Bit` comme suggéré dans les précédents rapports), respecte tout à fait la conception que nous avons proposée. Elle est constituée comme toutes les autres applications d'un service, et d'un widget `BitPlaneWindow` héritant de `StandardImageWindow` qui ajoute une barre d'outils (le widget `BitPlaneChoice`) permettant de sélectionner au moyen de cases à cocher les bits que l'on souhaite masquer dans l'image.

2.3.2 Filtrage

Mis à part le nom de certaines classes, la conception de cette application a été respectée. La concordance entre les noms initiaux et les noms réels est donnée ci-après :

- `FiltrageService` -> `FilteringService`
- `FiltrageWidget` -> `FilterChoice`
- `FiltrageAlgorithm` -> `Filtering`
- `SavedFilterWidget` -> `FilterEdition`

2.3.3 Binarisation

Le développement de l'application de Binarisation aura presque parfaitement suivi la conception. Le seul petit changement est l'utilisation directe d'un widget `HistogramWindow` au lieu de redéfinir un widget d'histogramme s'y basant qui n'aurait pas ajouté de fonctionnalité.

2.3.4 Étiquetage des composantes connexes

La conception de cette application est très proche de ce qui a effectivement été implémenté. Le diagramme de séquences utilisé pour décrire l'utilisation classique de l'application est complètement respecté, bien que les noms de classes aient changé depuis la conception.

Les deux widgets mentionnés ont été réalisés, avec des limites dans le cas du résultat : comme précisé pour la spécification, seuls le nombre de composantes et leur taille moyenne sont disponibles comme résultats de l'algorithme sur toutes les fonctionnalités proposées lors du rapport de conception.

De la même manière que pour les spécifications, les fonctionnalités additionnelles (modification de l'image en fonction des résultats) n'ont pas été implémentées par manque de temps.

2.3.5 Morphologie mathématique

Un certain nombre de modifications ont été effectuées sur la conception de cette application :

Fonctionnalités

La fonctionnalité de construction d'opérateur par combinaison a été supprimée, car très coûteuse et peu pertinente. En revanche beaucoup plus d'opérateurs prédéfinis ont été implémentés.

Architecture

L'application s'articule toujours sur la classe `MorphoMatService` (initialement `MorphoService`). En revanche, le nombre d'opérateurs ayant été augmenté, le service n'ajoute plus un bouton par opérateur à la barre d'outils, mais simplement deux boutons pour les opérateurs élémentaires (érosion, dilatation) et tous les autres dans un menu. De plus la fonctionnalité de combinaison d'opérateurs ayant été abandonnée, le widget `CombinationWidget` n'a plus de raison d'être. Le widget `StructuringElementWidget`, permettant de modifier l'élément structurant a quant à lui été découpé en deux widgets : `StructElemViewer` permettant de visualiser un élément structurant sous forme d'une grille de pixels et `StructElemWindow` rassemblant toutes les fonctionnalités d'éditeurs d'éléments structurants.

Pour tenir nos spécifications, nous avons parfois dû changer quelques éléments de notre conception. Néanmoins, ces changements restent peu importants. La lecture de notre rapport de conception est largement suffisante pour comprendre comment fonctionnent nos couches logicielles.

Chapitre 3

Phases de tests

Dans un projet de la taille de DETIQ-T, les tests peuvent être très utiles. Ils permettent, par exemple, de s'assurer que nos composants sont pleinement fonctionnels. Ainsi, nous pouvons savoir rapidement si la modification d'une implémentation en cours de développement entraîne un dysfonctionnement dans les cas les plus courants. Néanmoins, rédiger des tests unitaires utiles et pertinents n'est pas une tâche facile. Il faut savoir discerner les cas révélateurs. Pour mieux organiser nos tests, nous avons divisé le problème en deux parties : d'une part les tests de la bibliothèque **ImageIn** et d'autre part les tests de l'interface générique et des applications. Nous vous décrirons d'abord les tests d'**ImageIn** pour lesquels nous avons notamment développé un environnement de test automatique. Puis nous vous présenterons les tests de l'interface générique que nous avons basés sur le module *QTestLib*¹. Ce module est particulièrement adapté pour mettre en oeuvre des tests d'applications Qt, en fournissant un cadre de développement et un ensemble de macros et fonctions très utiles qui permettent de gagner énormément de temps.

3.1 Tests de IMAGEIN

Nous avons effectué deux types de tests sur la bibliothèque **ImageIn**. D'une part les tests « manuels » et d'autre part les tests « automatiques ».

3.1.1 Tests manuels

Les tests manuels que nous avons effectués consistent à appliquer manuellement une procédure (méthode ou algorithme de la bibliothèque) sur un jeu de données existant et arbitraire (ensemble de fichiers, ensemble d'images, etc.). La vérification des résultats se fait dans ce cas manuellement : validation visuelle puis vérification pixel par pixel d'une partie des résultats. L'ensemble des algorithmes et des fonctionnalités de la bibliothèque **ImageIn** a fait l'objet

1. <http://qt-project.org/doc/qt-4.8/qtestlib-tutorial.html>

de nombreux tests manuels sur des ensembles de données nombreux et variés. Les algorithmes ont par exemple tous été testés sur plusieurs dizaines d’images, présentant chacune des propriétés différentes (dimensions, nombre de canaux, etc.).

3.1.2 Tests automatiques

Les tests automatiques consistent quant à eux à appliquer automatiquement un ensemble de procédures de tests (mettant en œuvre des méthodes et des services de la bibliothèque) sur un jeu de données existant et/ou généré à la volée. Afin de mettre en œuvre ces procédures de tests automatiques, nous avons conçu un ensemble de classes dédié à cet effet.

Modélisation des tests

Chaque test unitaire est représenté par la classe abstraite **UnitTest**. Un objet **UnitTest** est un foncteur qui consiste à appliquer successivement l’initialisation du test, la procédure de test en elle-même et enfin la procédure de nettoyage (libération mémoire par exemple). Chaque **UnitTest** possède un nom permettant à l’utilisateur de l’identifier visuellement dans la sortie standard. Un ensemble de tests est ensuite représenté grâce à la classe abstraite **Tester**, comme décrit sur le diagramme UML de la Figure 3.1.

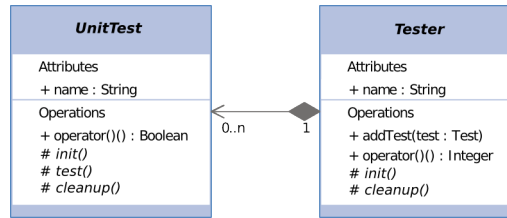


FIGURE 3.1 – Diagramme UML des classes **UnitTest** et **Tester**

Un **Tester** consiste à tester une fonctionnalité en appliquant successivement un ensemble de **Test**. Notre application de tests automatiques n’est donc finalement composée que d’un ensemble de **Tester**, chacun dédié au test d’une fonctionnalité. Chacune de ces différentes implémentations de **Tester** mettent en œuvre leurs propres **UnitTest**. Voici la liste des **Tester** utilisés :

- **CoreTester** teste le cœur de la bibliothèque
- **IOTester** teste les entrées/sorties fichiers.
- **ConverterTester** teste le convertisseur de type.
- **UtilityTester** teste les algorithmes « utilitaires »
- **MorphoMatTester** teste les algorithmes de morphologie mathématique
- **BinarizationTester** teste les algorithmes de binarisation

- **ComponentLabelingTester** teste les algorithmes d'étiquetage des composantes connexes
- **FilteringTester** teste les algorithmes de filtrage

CoreTester

CoreTester met en œuvre les **UnitTest** suivants :

- **CopyTest** teste les constructeurs de recopie/d'affectation
- **CropTest** teste la méthode de recadrage *Crop*
- **HistogramTest** teste le calcul d'histogramme
- **ProjHistTest** teste le calcul d'histogramme de projection

IOTester

IOTester ne met en œuvre qu'une seule procédure de tests générique, **IOTest**, qu'elle applique successivement pour les formats BMP, JPEG et PNG.

ConverterTester

ConverterTester ne met en œuvre elle aussi qu'une seule procédure de tests générique, **ConverterTest**, qu'elle applique successivement pour les types **Image_t**, **GrayscaleImage_t** et **RgbImage_t**.

UtilityTester

UtilityTester utilise une procédure de tests générique d'algorithmes, **AlgorithmTest**, qui consiste à appliquer un algorithme sur une image donnée et à comparer le résultat avec une autre image donnée. Pour effectuer cette comparaison nous avons créé la classe **ImageDiff**, permettant de mesurer la variation entre deux images en mesurant les distances entre pixels. Cela permet d'effectuer des comparaisons exactes (images identiques) ou approximatives (en acceptant un offset de variation, dans le cas de compression ou d'algorithmes non déterministes). **UtilityTester** met donc ainsi en œuvre **AlgorithmTest** afin de tester les algorithmes utilitaires (inversion, différence, moyenne, etc.) mais elle met aussi en œuvre une procédure de tests spécifique pour tester l'algorithme de plans de bits : **BitPlanTest**.

MorphoMatTester, BinarizationTester

Les **Tester** **MorphoMatTester** et **BinarizationTester** utilisent **AlgorithmTest** de la même manière que **UtilityTester**, afin de tester les différents algorithmes de morphologie mathématique et de binarisation.

ComponentLabelingTester

ComponentLabelingTester met en œuvre une procédure spécifique afin de tester l'algorithme de composantes connexes. Cette procédure de tests, **ComponentLabelingTest**, ne se base pas sur l'image de retour de l'algorithme (image colorée) mais sur les données relatives aux différentes classes déterminées par l'algorithme.

FilteringTester

De par la spécificité des algorithmes de filtrage (traitement dans un ensemble d'entiers relatifs puis conversion dans un ensemble naturel), **FilteringTester** met en œuvre une procédure de tests semblable à **AlgorithmTest** mais spécifique aux algorithmes de filtrage : **FilteringTest**.

Vous trouverez en Figure Figure 3.2 le diagramme complet décrivant la modélisation de l'utilitaire de tests automatiques.

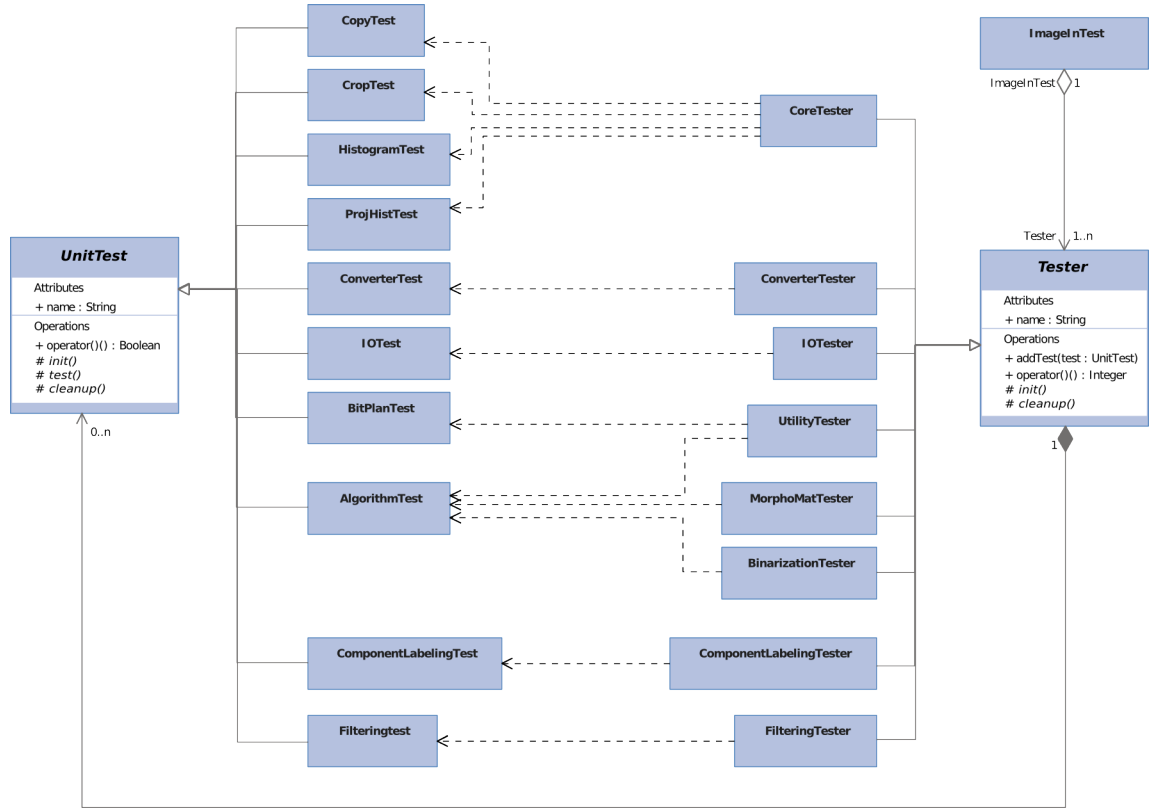


FIGURE 3.2 – Diagramme UML complet de l'utilitaire de tests automatiques

Utilitaire de tests

L'utilitaire de tests **ImageInTest** consiste uniquement à instancier et à appliquer les différents **Tester** décrits précédemment. Son code source (Figure 3.3) est particulièrement concis, la grande majorité du code étant réparti dans les différentes implémentations de l'interface **Test**. Cette architecture permet

donc une maintenance aisée de chacune des procédures de tests mis en œuvre ainsi qu'une gestion simple de l'ensemble du processus de tests. Vous trouverez Figure 3.4 un exemple de la sortie standard de l'utilitaire de tests.

```
Image_t<D>* generateRefImg();

int main(int argc, char** argv) {

    Image_t<D> *refImg = generateRefImg();

    unsigned int error = 0;

    error += CoreTester<D>(refImg)();
    error += IOTester<D>(refImg)();
    error += ConverterTester<D>()();
    error += UtilityTester(refImg)();
    error += MorphoMatTester()();
    error += BinarizationTester()();
    error += ComponentLabelingTester()();
    error += FilteringTester()();

    delete refImg;

    return error;
}
```

FIGURE 3.3 – Code source de l'utilitaire de tests

3.2 Tests de l'interface générique

Tester une interface graphique n'est pas une chose aisée. Contrairement aux fonctions et classes « classiques », l'implémentation d'une IHM est souvent peu « communicante ». C'est d'autant plus vrai que nous avons eu tendance à concevoir l'interface générique comme une boîte noire.

La phase de tests de notre interface générique a révélé que nous n'avions pas respecté pleinement l'architecture que nous avons décidé pour **GenericInterface**, lors de la création de nos widgets. Au final, ces derniers qui sont de véritables mini-contrôleurs (si on veut faire une parenthèse avec le modèle MVC²) alors qu'ils ne devraient avoir qu'un rôle de présentation, et déléguer les traitements à un service³ spécifique. Les tests *automatisés* auraient

2. Modèle-Vue-Contrôleur

3. Voir le manuel d'utilisation en annexe pour plus de précisions

été facilités et plus fiables. Cela ne les empêche pas d'être parfaitement fonctionnels et d'avoir été abondamment testés « manuellement ». Force est cependant de constater qu'ils sont sans doute moins évolutifs et maintenables que nous l'avions imaginé.

Comme les Widgets étaient difficilement testables de façon rigoureuses, nous nous sommes concentrés sur les Services. Les fonctionnalités testées sont, par exemple :

- WindowService : ouverture d'un fichier ;
ouverture d'une image, vérification de la réussite, vérification des paths
- FileService : enregistrement d'une image ;
ouverture d'une image, enregistrement sous un nom différent, comparaison des caractéristiques principales
- StandardImageWindow : ouverture d'un histogramme ;
ouverture d'une image, création d'un histogramme, vérification de la réussite

3.3 Tests des applications

Nos applications se basent toutes en grande partie sur l'interface générique. De ce fait, la majorité de leurs fonctionnalités (elles sont en effet très simples et reposent en grande partie sur le comportement de base induit par l'utilisation de l'interface générique) n'a pas besoin d'être retesté.

Pour la plupart, les applications n'embarquent qu'un Service supplémentaire, une classe qui hérite de `AlgorithmService`. La procédure de tests retenue est simple : appliquer un algorithme avec et sans l'interface générique et comparer les résultats. Si tout se passe bien, ils devraient être identiques. Néanmoins, avoir deux images parfaitement identiques est une utopie. Aussi, nous ne testons pas la stricte égalité (qui ferait échouer les tests systématiquement) mais que la différence entre chaque composante de chaque pixel est inférieure à un delta arbitraire.

À l'instar de l'interface générique — ce qui n'est guère surprenant, étant donné qu'elles sont construites sur son modèle — les applications pédagogiques sont difficilement testables absolument. Elles ont été abondamment utilisées, en multipliant les mises en situation, afin d'essayer de trouver les *bugs*, mais de façon automatisée, c'est une tâche difficile qui aurait dû être réfléchie dès la conception.

```
zakinster@ubuntu:~/dev/detiqt/src/Tests/ImageIn$ ./ImageIn_test
Testing ImageIn Core
    Copy constructor...Success
    Image crop...Success
    Histogram...Success
    Projection histogram...Success

Testing I/O File
    BMP I/O...Success
    JPEG I/O...Success (Image diff = min:0, max:4, mean:0.309996)
    PNG I/O...Success

Testing Image converter
    GrayscaleImage <-> RgbImage...Success
    RgbImage <-> Image...Success
    GrayscaleImage <-> Image...Success

Testing Utility algorithms
    Identity...Success
    Inversion...Success
    Difference...Success
    Average...Success
    BitPlane...Success

Testing Mathematical morphology
    Erosion d15 on M...Success
    Dilatation d15 on M...Success
    Opening d15 on M...Success
    Closing d15 on M...Success
    Gradient d3 on M...Success
    White Top Hat d15 on M...Success
    Black Top Hat d15 on M...Success
    Gradient d3 on rose...Success
    Gradient d3 on lena...Success

Testing Binarization
    Otsu on harewood...Success
    Otsu on snow...Success
    Otsu on nutsBolts...Success

Testing Component labeling
    Rice (w/ open-close)...Success
    Rice & sugar...Success
    M (w/ white top-hat)...Success
    QR Code...Success

Testing Filtering
    Linear Blur...Success
    Gaussian blur...Success
    Prewitt...Failed (Image diff = min:0, max:255, mean:93.9432)
```

FIGURE 3.4 – Exemple de sortie de l'utilitaire de tests

Chapitre 4

Finalisation du projet

Avoir une vision claire de l'état de finalisation de notre projet est essentiel pour quiconque désire le reprendre. En nous attardant sur chaque composant de DETIQ-T, nous allons nous attacher à résumer ce qui est fait, ce qu'il reste à faire pour coller aux spécifications initiales et nous essaierons de donner, quand il y en a, des pistes d'approfondissement.

4.1 Bibliothèque IMAGEIN

Tous les objectifs que nous nous étions fixés pour la bibliothèque IMAGEIN ont été réalisés. La structure de base permet de répondre à tous les besoins des applications pédagogiques, et même plus encore. Beaucoup d'algorithmes divers et variés ont été implémentés, et l'extensibilité est bien présente. Il y a cependant quelques points qui pourraient bénéficier de plus de temps de développement.

Extensibilité Les possibilités de la bibliothèque ImageIn, de par sa nature générique, dépassent le cadre de nos applications. Si la bibliothèque a été intensément testée et utilisée dans le cas des Images dont les pixels sont représentés par un ensemble d'entiers sur 8 bits, son utilisation avec d'autres types d'**Image** (profondeur d'image supérieure à 8 bits par canal ou système colorimétrique non entier) est théoriquement possible mais peut nécessiter quelques ajustements. Notre objectif n'était cependant pas de développer directement une bibliothèque universelle mais plutôt une bibliothèque très facilement extensible.

Simplicité d'utilisation L'aspect générique de la bibliothèque, indispensable à son extensibilité, ne facilite pas sa prise en main. En effet, l'utilisation intensive de template et d'héritage, jusqu'à jouer avec les limites du langage C++, ne facilite pas la tâche de l'utilisateur quand il s'agit de comprendre le fonctionnement interne de la bibliothèque. Beaucoup d'efforts ont cependant été faits pour clarifier l'interface, simplifier l'utilisation sans pour autant nuire à l'extensibilité de la bibliothèque. Ces simplifications qui ont été faites, notamment

à l'aide de nombreux *typedef* et de spécialisations partielles de templates, ont beaucoup aidé à la simplicité d'utilisation mais n'ont pas été poussées jusqu'au bout. Il est donc encore possible de clarifier l'interface d'utilisation, notamment au niveau de la couche Algorithme.

4.2 Interface générique

Nous désirions donner un moyen simple d'utiliser `ImageIn` afin de construire des applications graphiques complètes. Le choix d'utiliser Qt nous a permis de conserver la portabilité de notre bibliothèque. De plus, le *framework* est réputé assez simple à utiliser et dispose d'une communauté et d'un support important.

En ayant fait le choix d'enrichir l'interface grâce à des Services plutôt qu'en encourageant l'implémentation d'une classe abstraite comme nous l'avions un temps imaginé, nous avons pu créer un système favorisant la réutilisation du code. À titre d'exemple, il serait très simple de créer une application regroupant toutes les fonctionnalités de nos applications en une seule. Il suffit de faire un appel à la fonction `addService` de `GenericInterface`.

La plupart de nos objectifs ont été atteints, cela ne veut cependant pas dire que l'interface générique est terminée. Pour l'heure, il s'agit d'une fonction minimaliste et nous pouvons facilement dégager deux axes de travail à poursuivre :

Soigner la base existante — l'interface générique est un projet assez important qui tire parti le plus possible du système de signaux et de slots de Qt. Ainsi les connexions qui existent entre les différents composants de l'interface ne sont pas toujours évidentes, surtout que nous avons eu tendance à les déclarer un peu partout dans le code. Afin de pérenniser son extension, il serait sans doute profitable de les regrouper dans une fonction particulière à chaque fois, à la façon de la fonction `connect` de `GenericInterface`. Cette étape réalisée, il pourrait être utile d'améliorer et d'optimiser les Widgets existants. Ils fonctionnent, font ce qu'on leur demande mais il serait facile d'imaginer d'autres fonctionnalités intéressantes, un affichage plus intuitif, etc.

Proposer des Services facultatifs — la version actuelle de l'interface générique est minimaliste et il est important qu'elle le reste : c'est son but. Elle doit implémenter des outils de base mais n'apporter rien d'inutile ou de superflu. Néanmoins, cela n'empêche pas que le système des Services est spécialement prévu pour permettre aux contributeurs éventuels — rappelons que DETIQ-T dans son ensemble est un projet libre — de proposer des fonctionnalités supplémentaires. On aurait alors quelque chose proche des *Bundle* de *Symfony2*¹, qui sont de véritables briques logicielles.

1. Un framework PHP... français !

Les pistes ne manquent donc pas pour continuer le développement de `GenericInterface` et de ses composants.

4.3 Applications pédagogiques

DETIQ-T prévoyait la réalisation d'applications de traitement d'images. Sur les six prévues, nous en avons implémenté cinq ; la dernière (qui devait permettre d'utiliser un algorithme de projection) n'est cependant pas totalement abandonnée car les histogrammes de projection ont été intégrés dans l'interface générique (voir l'annexe pour l'utilisation de l'interface générique).

4.3.1 Application de masques binaires

L'application utilisant les masques binaires *BitPlane* est une application très simple qui remplit déjà parfaitement son cahier des charges. On peut la considérer comme étant terminée.

4.3.2 Filtrage

L'application délivrée est terminée : elle implémente les principaux algorithmes de filtrage comme annoncé et permet en plus de créer soi-même ses propres filtres, afin de tester et de mieux comprendre le fonctionnement de cette technique de traitement d'images.

Néanmoins, on peut remarquer que dès que le filtre atteint une certaine taille, le temps de traitement devient important, voir rédhibitoire. Il serait envisageable, pour certains filtres parmi les plus utilisés, de ne pas utiliser le produit de convolution comme c'est fait actuellement mais de regarder du côté des transformées de Fourier. Le filtre gaussien, entre autre, se prête très bien à ce type d'exercice et la complexité de son application devient indépendante du rayon (dans le cas actuel, plus le rayon est grand, plus la matrice est grande). Il existe de très nombreux documents sur les transformées de Fourier d'une part, et sur la façon dont on peut appliquer le flou gaussien grâce à elles d'autre part. L'astuce est que la matrice gaussienne repose elle-même sur une fonction qui possède sa propre transformée. Il suffit ensuite de multiplier composante par composante les deux matrices obtenues et d'appliquer une transformée inverse.

Il s'agit d'une piste sérieuse d'amélioration de l'algorithme (afin de le rendre plus efficace).

4.3.3 Binarisation

Pour le moment, l'application de binarisation n'implémente que deux méthodes de choix de seuil : soit manuellement (avec un ou deux seuils) soit grâce

à l'algorithme d'Otsu. Il pourrait être intéressant d'implémenter d'autres façons de faire. Par exemple, il existe des méthodes utilisant des seuils variables, afin d'obtenir un meilleur affichage final. Ce genre de techniques de traitement d'images aurait parfaitement sa place dans cette application.

4.3.4 Étiquetage des composantes connexes

L'application d'étiquetage des composantes connexes offre déjà une base fonctionnelle pour observer les effets de cet algorithme avec différents paramètres. Cependant, il est possible d'améliorer l'intérêt de cette application grâce à des fonctionnalités secondaires qui ont été mentionnées dans les précédents rapports.

Tout d'abord, il serait intéressant d'obtenir plus d'informations sur les composantes connexes identifiées. Pour l'instant, seuls la taille et le nombre de composantes sont renseignés après l'application de l'algorithme. Nous pourrions offrir la possibilité d'en fournir d'autres (espacement entre les composantes, écart type pour ces différentes mesures...)

Enfin, un objectif à plus long terme peut être de pouvoir modifier l'image en fonction du résultat, comme mentionné dans le rapport de conception (supprimer les composantes les plus petites ou fusionner les plus proches). Ces modifications permettraient d'identifier des intérêts concrets à cet algorithme.

4.3.5 Morphologie mathématique

L'application de morphologie mathématique offre déjà un nombre important d'algorithmes, et une grande liberté quant au choix de l'élément structurant. Certains algorithmes intéressants n'ont cependant pas été implémentés tels que les opérateurs de reconstruction (ouverture et fermeture itératives jusqu'à stabilité) ou de segmentation. Il est donc encore possible d'enrichir de manière assez simple le panel d'algorithmes de cette application. De plus, si l'utilitaire de choix d'élément structurant offre déjà une grande liberté à l'utilisateur en étant capable d'ouvrir un fichier image comme élément structurant en plus des éléments prédéfinis, il pourrait être intéressant d'offrir la possibilité de l'éditer directement dans l'application via un outil d'édition rudimentaire.

Conclusion

Le projet DETIQ-T est désormais un projet qui peut facilement s'inscrire sur le long terme, c'est d'ailleurs ce qui nous a poussé à l'envisager, dès le départ, comme un projet libre. Il est d'ailleurs destiné à être publié sous licence LGPL. Aussi, il est normal que le présent rapport donne un nombre important de pistes pour étoffer le projet. Actuellement, DETIQ-T n'est pas encore assez mature pour prétendre à une version « stable ». Néanmoins, il présente un état d'avancement suffisant pour être fonctionnel et permettre de réaliser des applications qui se basent dessus ; les applications pédagogiques présentées dans ce document le prouvent.

Les spécifications et la conception d'origine ont été en grande partie respectées, ce qui donne au projet une lisibilité importante. Heureusement, d'ailleurs, car si **ImageIn** comme **GenericInterface** ont été pensés pour être le plus facilement utilisables possible, ils n'en restent pas moins des projets importants ; notre volonté de proposer une solution générique et extensible est à la fois leur force — ils sont susceptibles d'être utilisables dans beaucoup de cas différents — et leur faiblesse. La généricité se fait forcément au détriment de la simplicité. Il est donc primordial d'avoir une ligne directrice claire et cela a été notre cas pendant ces mois de développement.

Au final, nous espérons livrer un projet capable de susciter de l'intérêt, que ce soit par le biais d'un nouveau projet INSALien ou en parvenant à le diffuser dans les sphères du logiciel libre.

Annexe A

Manuel d'utilisation

A.1 Bibliothèque IMAGEIN

A.1.1 Utilisation de la bibliothèque

La bibliothèque IMAGEIN fournit un ensemble de fonctionnalités de base permettant de manipuler les images matricielles et d'y appliquer des algorithmes. L'ensemble de la documentation de ces classes est disponible à l'adresse <http://d.3f0.net/doc/ImageIn/html> dans sa version la plus à jour. Ce document permet cependant de mieux appréhender l'utilisation de cette bibliothèque pour quelqu'un n'ayant aucune connaissance de sa structure.

La plupart des classes de la bibliothèque disposent d'un ou plusieurs paramètres templates, et leurs noms présentent le suffixe `_t`. Cependant, il existe des typedef permettant d'ignorer ce paramètre template et d'utiliser la même classe sans ce suffixe. L'utilisation de ce paramètre template sera détaillée dans la section « Fonctionnalités avancées ».

Manipulation des images

La gestion des images se fait via la classe `Image`, ou de manière plus spécifique par l'une des classes spécialisées `GrayscaleImage` ou `RgbImage`. La classe `Image` représente en fait une matrice à trois dimensions : largeur, hauteur et nombre de canaux de l'image. Toutes les images matricielles peuvent donc être représentées par cette classe.

Cette classe peut être construite de deux manières différentes :

- à partir des dimensions de l'image et d'un tableau de `uint8_t` ;
- à partir d'un nom de fichier (formats supportés : png, jpg, png).

Elle dispose également des accesseurs standards `getWidth()`, `getHeight()`, `getNbChannels()` ainsi que d'une fonction `getPixel()` et `setPixel()` dont l'utilisation est très simple et naturelle. On notera également la présence d'une méthode `save()` permettant l'enregistrement de l'image dans un fichier (les formats supportés sont les mêmes que pour l'ouverture).

L'exemple de code suivant donne un aperçu des fonctionnalités en question. Il ouvre une image, trace une croix noire sur ses diagonales (si l'image est rectangulaire, il s'agira des diagonales du carré de côté égal à la largeur de l'image) et l'enregistre en écrasant l'image d'origine.

```
#include <Image.h>

using namespace imagein;

int main()
{
    Image im("samples/test.jpg");

    for(int i = 0 ; i < im.getWidth()
        && i < im.getHeight() ; ++i) {
        for(int j = 0 ; j < im.getNbChannels() ; ++j) {
            im.setPixel(i, i, j, 0);
            im.setPixel(im.getWidth()-i-1, i, j, 0);
        }
    }

    im.save("samples/test.jpg");
}
```

Les deux classes spécialisées `GrayscaleImage` et `RgbImage` n'apportent pas de fonctionnalités supplémentaires à cette classe. Il s'agit uniquement de simplification de l'interface pour des images à un ou trois canaux. Il devient donc inutile de préciser le nombre de canaux à la construction ou aux accès d'un pixel. De même, l'ouverture d'un fichier dont le nombre de canaux ne correspond pas lèvera une exception. L'ensemble des accesseurs disponibles pour ces classes est disponible dans la documentation.

Appliquer un algorithme

En plus de manipuler les images directement, `ImageIn` permet d'appliquer des algorithmes sur ces images. Ces algorithmes se présentent tous sous forme d'objets foncteurs, et prennent parfois des paramètres à la construction. Ils sont ensuite appliqués via l'opérateur d'application `()`. La plupart des algorithmes prennent une seule image en paramètre, mais d'autres peuvent en prendre deux ou plus. Ils renvoient un pointeur sur une nouvelle image nouvellement allouée.

Le type d'image en entrée est spécifié dans la documentation. Si l'image passée en paramètre n'est pas du bon type, une exception sera levée. Pour garantir le bon type, les images peuvent être converties via la classe **Converter** qui sera détaillée dans la section correspondante.

L'exemple de code suivant applique un algorithme de binarisation (Otsu) à une image et enregistre l'image binarisée en écrasant l'image d'entrée.

```
#include <Image.h>
#include <Algorithm/Otsu.h>

using namespace imagein;
using namespace imagein::algorithm;

int main()
{
    GrayscaleImage im("samples/test.jpg");
    Otsu o;

    Image* im_binaire = o(&im);
    im_binaire->save("samples/test.jpg");
    delete im_binaire;
}
```

La classe Converter

Dans la réalité, la plupart des fichiers images sont en fait des images RGB, même si elles représentent des images en niveau de gris (trois canaux égaux). En conséquence, il peut être gênant d'utiliser des algorithmes demandant une image en niveau de gris en entrée. De la même manière, on peut vouloir convertir une **GrayscaleImage** en **RgbImage** pour utiliser un algorithme demandant ce type d'image en entrée. On peut également vouloir revenir à la classe **Image**, plus générique à partir d'une de ses sous-classes.

C'est pour répondre à ce besoin que la classe **Converter** est utile. Cette classe est templâtée par le type de destination de la conversion, et dispose d'une méthode surchargée pour chaque type d'image source. Cette méthode est statique, la classe **Converter** n'a donc jamais besoin d'être instanciée. Un exemple d'utilisation est donné par le code suivant (identique au précédent mais en utilisant un convertisseur).

```
#include <Image.h>
#include <Algorithm/Otsu.h>
#include <Converter.h>

using namespace imagein;
```

```
using namespace imagein::algorithm;

int main()
{
    Image im("samples/test.jpg");
    Otsu o;

    Image* im_binaire = o(Converter<GrayscaleImage>::convert(im));
    im_binaire->save("samples/test.jpg");
    delete im_binaire;
}
```

L'ensemble des méthodes de conversion peut être consulté dans la documentation de la bibliothèque.

A.1.2 Fonctionnalités avancées

Gestion des images de différentes profondeurs

La profondeur d'une image représente le nombre de bits utilisés pour la valeur d'un pixel. Par exemple, une image RGB classique a une profondeur de 8 bits : chaque canal prend sa valeur dans l'intervalle $[0, 255]$. Dans IMAGEIN, il est possible de gérer des images de différentes profondeurs via le paramètre template des différentes classes.

Par exemple, il est possible de créer une image de profondeur 32 bits par une instance de la classe `Image_t<depth32_t>` (`depth32_t` est un typedef vers un entier non signé sur 32 bits). Des typedefs sont fournis pour les profondeurs 8, 16 et 32 bits. De plus, il existe un typedef `depth_default_t` pour la profondeur par défaut (celle qui est utilisée pour les classes sans paramètre template, détaillées dans la section précédente).

La contrainte liée à ces profondeurs est assez importante. En effet, un algorithme prenant en paramètre une image d'une certaine profondeur devra rendre une image de cette même profondeur en résultat, les algorithmes étant eux-mêmes liés à un paramètre template.

Au delà des types de retours ou des paramètres, utiliser le paramètre template des classes n'a pas d'influence sur l'utilisation de cette classe.

Les classes `ImageFile`

Les classes `ImageFile` sont en fait un ensemble de classes gérant l'ouverture et la fermeture des fichiers images. L'interface de ces classes est de plus bas niveau que les classes images, et leur utilisation est réservée à des fins bien spécifiques.

Il existe une classe pour chaque format de fichier, cependant elles implémentent toutes la même interface. Ces classes sont créées à partir d'un nom de fichier et permettent plusieurs opérations basiques :

- lire la hauteur, la profondeur et le nombre de canaux d'une image ;
- lire la profondeur d'une image (résultat en bits) ;
- lire et écrire les données dans le fichier.

Contrairement aux classes de plus haut niveau, la profondeur n'est pas gérée directement. Les données sont toujours sous la forme de pointeurs sur `void`. C'est à l'utilisateur de savoir quelle profondeur le fichier utilise et de manipuler ce tableau en conséquence.

La classe étant différente selon le format de fichier considéré, il est possible de récupérer une instance de la bonne classe via une fabrique abstraite en utilisant le code suivant :

```
ImageFile* imf = ImageFileAbsFactory::getFactory()->getImageFile(filename);
```

A.1.3 Étendre la bibliothèque

La philosophie de IMAGEIN est d'être facilement extensible. Ainsi, il est possible d'ajouter de nouveaux formats de fichiers ou de nouveaux algorithmes facilement, et ce sans recompiler la bibliothèque. Bien que cela soit moins utile, il est également possible d'ajouter de nouveaux types d'images.

Ajouter ou modifier le support d'un format de fichier

Le support des formats de fichiers est fait via les classes `ImageFile` et l'ajout d'un nouveau format est fait en trois étapes :

1. Créer une classe héritant de `ImageFile` qui gère ce format de fichier ;
2. Créer une classe héritant de `ImageFileFactory` capable d'instancier la classe créée à l'étape 1 ;
3. Prendre en compte cette nouvelle fabrique dans la fabrique abstraite.

La première étape est assez simple puisqu'il suffit d'implémenter les fonctions virtuelles pures de `ImageFile`. La façon d'implémenter ces fonctions dépend fortement du format de fichier géré. Il est recommandé d'utiliser des bibliothèques le plus standard possible pour des raisons de performances.

Les deuxième et troisième étapes sont nécessaires pour que ce format d'image soit pris en compte dans le reste de la bibliothèque. Ces étapes sont détaillées via l'exemple de code suivant (on suppose que la classe créée à l'étape 1 s'appelle `GifImage` :

```

#include "GifImage.h"
#include <ImageFileFactory.h>

using namespace imagein;

class MyImageFileFactory : public ImageFileFactory
{
public:
    ImageFile* getImageFile(std::string filename)
    {
        if(filename.substr(filename.size()-4,4)==".gif") {
            return new GifImage(filename);
        }
        else
            //inutile de coder a nouveau le comportement deja
            //gere par la bibliotheque.
            return this->ImageFileFactory::getImageFile(filename);
    }

    //Il n'est pas necessaire de recoder la methode
    //getImageDepth, car elle fait appel a la methode
    //getImageFile.
};

int main()
{
    //prise en compte de la nouvelle factory
    ImageFileAbsFactory::setFactory(new MyImageFileFactory);

    //dans la suite du programme, le format gif sera pris
    //en compte puisque toute la bibliotheque utilisera la
    //nouvelle factory.
}

```

Créer un algorithme

Il est également très simple de créer un nouvel algorithme. En effet, la classe `Algorithm_t` (et d'autres classes dont on ne parlera pas ici) sont développées de manière à permettre l'implémentation d'un algorithme en redéfinissant uniquement la méthode correspondant à l'algorithme lui-même. Les différentes surcharges de l'opérateur d'application sont déjà développées, et l'utilisateur n'a pas à s'en soucier.

Il convient cependant d'hériter de la bonne classe : `Algorithm_t`. En effet, cette classe dispose de paramètres templates en fonction de l'arité de l'algorithme et de son type de retour. Par exemple, pour créer un algorithme

d'arité 2 retournant une image en niveau de gris, il faut hériter de la classe `Algorithm_t<GrayscaleImage, 2>`. Notez que cet algorithme travaillera bien sur des `GrayscaleImage_t` de profondeur 8 bits. Pour d'autres profondeurs, il aurait fallu hériter de `Algorithm_t<GrayscaleImage_t<D>, 2>` par exemple.

Une fois la classe du nouvel algorithme créé, il suffit de redéfinir la méthode `GrayscaleImage* algorithm(const std::vector<const Image_t<D>*>& imgs)`. La signature de la méthode à surcharger est toujours la même, seul le type de retour change, et doit être un pointeur sur le type désigné par le premier paramètre template. La profondeur des images du vecteur doit également correspondre au type de retour (utiliser `depth_default_t` pour les classes sans paramètre template).

Dans le corps de la fonction, il conviendra de tester le type des images d'entrées (via un `dynamic_cast` par exemple). En revanche, il n'est pas nécessaire de tester la taille du vecteur : elle sera toujours égale à l'arité de l'algorithme.

Si votre algorithme a besoin d'autres paramètres que les images d'entrée, il conviendra de surcharger le constructeur pour mémoriser ces paramètres. Ils seront communs à chacune des applications de l'algorithme.

L'exemple suivant montre l'implémentation d'un algorithme qui fait la moyenne pondérée de deux images. Il fonctionne sur n'importe quel type d'image et renvoie donc une image générique.

```
template <typename D>
class MoyennePonderee : public Algorithm_t<Image_t<D>, 2>
{
public:
    MoyennePonderee(int a, int b) : _a(a), _b(b) {};

protected:
    Image_t<D>* algorithm(const std::vector<const Image_t<D>*>& imgs)
    {
        const GrayscaleImage_t<D>* img1;
        img1 = dynamic_cast<const GrayscaleImage_t<D>*>(imgs.at(0));
        const GrayscaleImage_t<D>* img2;
        img2 = dynamic_cast<const GrayscaleImage_t<D>*>(imgs.at(1));

        if(img1 == NULL || img2 == NULL) {
            throw std::exception();
        }

        //Dans un vrai algorithme, il faudrait tester l'egalite des
        //dimensions des deux images...
```

```

D* data;
data =
    new D[img1->getWidth() * img1->getHeight() * img1->getNbChannels()];
Image_t<D>* out;
out = new GrayscaleImage_t<D>(img1->getWidth(),
                               img1->getHeight(),
                               img1->getNbChannels(),
                               data);

for(int i = 0 ; i < img1->getWidth() ; ++i) {
    for(int j = 0 ; j < img1->getHeight() ; ++j) {
        for(int k = 0 ; k < img1->getNbChannels() ; ++k) {
            int value = a*img1->getPixel(i,j,k) + b*getPixel(i,j,k);
            value /= a+b;
            out->setPixel(i,j,k,value);
        }
    }
}

return out;
}

private:
    int _a, _b;
};

```

A.2 Interface générique

A.2.1 Comportement pré-défini

L'interface générique, telle qu'elle est implémentée actuellement, sert de base à toutes les applications pédagogiques que nous avons développées. Ce choix technique a pour but de donner à nos réalisations un comportement commun, afin de mettre en valeur leurs relations — elles dépendent toutes du même projet — et d'unifier leur manière de fonctionner.

Gestion des fenêtres ouvertes

Notre interface générique est ce que l'on appelle une *MDI Area*¹, ce qui veut dire qu'elle possède une zone de travail dans laquelle évoluent plusieurs documents sous forme de fenêtres.

Nous le verrons par la suite, l'utilisateur aura l'occasion d'avoir beaucoup de fenêtres ouvertes pour, par exemple, disposer d'informations supplémentaires

1. Multiple Document Interface

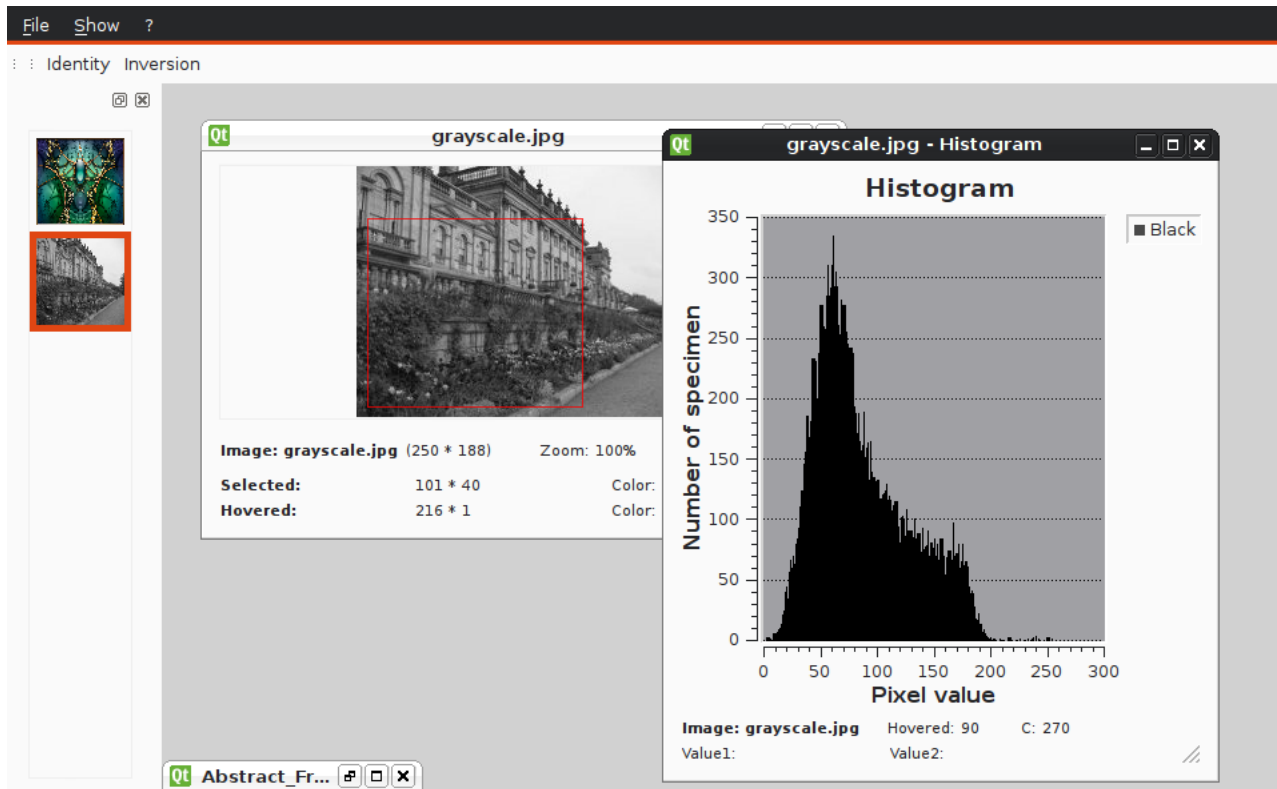


FIGURE A.1 – Vue d'ensemble de l'interface générique

sur une image en particulier. Cette multiplication d'éléments graphiques nous a poussé à réfléchir soigneusement à l'ergonomie que nous voulions proposer. La Figure A.1 donne une vue d'ensemble de notre interface générique. La barre de navigation, à gauche, est l'élément central qui régit la politique de réduction et d'agrandissement automatique des fenêtres. Son comportement est très simple :

- quand vous ouvrez une image (menu *Files*, *Open*), elle s'ajoute dans la barre de navigation ;
- quand une fenêtre est créée automatiquement, elle est liée à sa fenêtre d'origine ;
- quand vous cliquez sur une miniature dans la barre de navigation, toutes les fenêtres liées se mettent au premier plan, les autres se réduisent ;
- il est possible de faire une sélection multiple en maintenant la touche Contrôle (Ctrl) appuyée.

Outils pour l'analyse d'une image

Par défaut, l'interface générique de DETIQ-T fournit un certain nombre d'outils pour analyser en détails une image. Ils sont accessibles de deux façons : soit

par le biais d'un clic droit sur l'image, soit via le menu *Show*.

L'outil le plus utilisé est sans doute l'histogramme (*Histogram*). Il permet une visualisation par canaux qu'il est possible de superposer. Les histogrammes de projection horizontaux (respectivement verticaux) demandent une valeur numérique pour un canal et comptent le nombre d'occurrences de cette valeur pour ce canal sur chaque ligne (respectivement colonne) de l'image. Enfin, une grille de pixels (*Pixel Grid*) permet de visualiser distinctement les pixels de l'image. Chaque composante peut être vue individuellement (les valeurs s'affichent alors), deux à deux ou les trois en même temps.

À noter que les widgets liés aux histogrammes s'appliquent sur une sélection (cadre rouge sur l'image d'origine) qu'il est possible de modifier après coup : l'histogramme se met à jour dynamiquement ! Il est possible de modifier sa taille en cliquant sur le bord (le curseur change de forme). En pressant la touche Contrôle en même temps, il est possible de déplacer facilement la sélection.

A.2.2 Utilisation dans un projet

Le but de l'interface générique n'est pas d'être utilisée seule, son intérêt est d'avoir été spécifiquement pensée pour être facilement enrichie de nouveaux modules, nommés ici « Services ». Nous allons nous attacher à comprendre comment un développeur peut ajouter une fonctionnalité à cette interface.

Philosophie des services

Voici la définition d'un service, telle qu'elle est donnée par le site le-dictionnaire.com.

Service : *action de s'acquitter de certains devoirs, ou de certaines fonctions ; le résultat de cette action.*

Dans notre cas, les Services ne possèdent qu'un devoir : agir sans perturber le fonctionnement de l'interface générique. Ils possèdent aussi des fonctions, propres à chacun.

À chaque tâche son Service. Parce qu'un Service est une boîte noire et qu'on ne sait pas forcément comment il est implémenté, il est important de ne pas chercher à se substituer à son action. Il existe, par défaut, un Service qui se charge de la gestion des fichiers images. Si vous avez une tâche qui nécessite de sauvegarder une image, il faut la lui déléguer (nous verrons comment par la suite). Si deux de vos services font, à un moment, la même chose, c'est peut-être qu'il en faut un troisième (ou, au contraire, qu'il faut les fusionner).

À chaque Service sa tâche. Si chaque tâche ne doit être assurée que par un Service, il est aussi préférable de limiter un service à une tâche bien précise. Cela permet d'avoir des « briques » logicielles bien distinctes et plus facilement réutilisables dans un autre projet. Cette philosophie implique d'avoir beaucoup d'échanges de signaux entre les différents composants de l'interface, mais ce n'est pas très grave car cette fonctionnalité est très bien gérée par Qt. Il serait dommage de ne pas s'en servir.

Créer et utiliser un Service

Un service implémente l'interface `Service`, qui possède relativement peu de méthodes.

```
namespace genericinterface
{
    class Service
    {
    public:
        virtual void connect(GenericInterface* gi) =0;
        virtual void display(GenericInterface* gi) =0;
    };
}
```

À noter que vous ne devez pas utiliser le namespace `genericinterface`, qui est réservé pour les fonctions de l'interface générique seulement. N'hésitez pas à visiter la documentation de la classe `Service`².

La fonction `void connect(GenericInterface* gi)` permet d'interconnecter les services entre eux. Pour récupérer des pointeurs vers d'autres services, la `GenericInterface` fournit plusieurs outils, comme un `enum` :

```
namespace genericinterface {
    enum {
        WINDOW_SERVICE,
        FILE_SERVICE,
        UTILITY_SERVICE
    };
}
```

La fonction `void display(GenericInterface* gi)` permet au Service de s'intégrer à l'interface générique. Cela permet d'ajouter des menus, des docks, des icônes dans la barre des tâches... Étant donné que `GenericInterface` hérite de `QMdiArea`, elle s'utilise de la même façon. N'hésitez pas à regarder la documentation de Qt³, très bien faite, pour savoir comment l'utiliser de façon efficace.

2. http://d.3f0.net/doc/GenericInterface/html/classgenericinterface_1_1_service.html

3. <http://doc.qt.nokia.com/4.7-snapshot/index.html>

Le schéma typique pour lier deux services entre eux (dans la fonction connect) est le suivant :

```
void MyService::connect(GenericInterface* gi) {
    /* ... */
    MyOtherService* serv;
    serv = dynamic_cast<MyOtherService*>(_gi->getService(_id))
    QObject::connect(this, SIGNAL(mySlot),
                     serv, SLOT(oneSlot));
    /* ... */
}
```

Utiliser les services déjà existants

Il est très important de ne pas réinventer la roue, pour plusieurs raisons. Si l'interface générique fournit plusieurs services par défaut, ce n'est pas un hasard. Ils permettent d'assurer un comportement standard en terme d'ergonomie et c'est pourquoi il est si important d'utiliser, par exemple, le `WindowService` pour ajouter une fenêtre dans la *MDI Area*. En effet, c'est ce service qui s'assure de la cohérence entre les fenêtres, de leur réduction et de leur agrandissement au bon moment. Ajouter une fenêtre sans passer par le `WindowService`, c'est l'extraire de cette logique de fonctionnement, et donc perdre en ergonomie.

A.2.3 Créer un widget DETIQ-T

Créer un Service permet d'ajouter une fonctionnalité à l'interface générique, mais il n'est pas rare qu'il repose sur des Widgets, par exemple `ImageWindow`.

La manière la plus simple de procéder est d'hériter de cette classe. Ne perdez cependant pas de vue que l'interface générique donne déjà accès à plusieurs types hérités :

- `GridWindow`, pour un widget avec visionneur ;
- `HistogramWindow`, pour un widget contenant un histogramme ;
- `StandardImageWindow`, pour un widget contenant une image.

Il existe quelques autres types moins intéressants. Encore une fois, la documentation de `GenericInterface` est assez complète pour vous permettre de savoir quelles sont les méthodes à implémenter.

A.3 Compilation

DETIQ-T est un projet de taille conséquente, organisé en plusieurs couches. En conséquence de cette architecture, et des bibliothèques externes utilisées, la procédure de compilation n'est pas forcément évidente. Cette section du manuel d'utilisation a donc pour but d'expliquer la démarche de compilation du projet

DETIQ-T, de la bibliothèque IMAGEIN jusqu'à une application unitaire utilisant l'interface générique.

Cette compilation se fait en trois étapes :

1. Compilation de IMAGEIN
2. Compilation de l'interface générique
3. Compilation d'une application utilisant l'interface générique

A.3.1 Pré-requis

Ce document suppose que les sources sont enregistrées selon l'arborescence suivante :

- la bibliothèque IMAGEIN dans un dossier du même nom
- l'interface générique dans un dossier "GenericInterface"
- l'application unitaire dans un dossier "Application"
- ces trois dossiers cote à cote dans votre système de fichiers.

Tout le projet est écrit en C++ standard, et utilise uniquement des bibliothèques multi-plateformes. La procédure de compilation décrite dans la suite est la même pour Windows ou Unix et doit fonctionner pour ces deux systèmes. Pour Windows, cette procédure fonctionnera en utilisant le shell `msys` fourni avec les versions récentes de MinGW⁴.

Le projet utilise les bibliothèques externes suivantes :

- `libjpeg`, `libpng` pour l'ouverture/fermeture des images
- `Qt` et `Qwt` pour l'interface graphique.

Toutes ces bibliothèques sont multi-plateformes, et il est recommandé de les installer à la version la plus récente pour éviter tout problème.

A.3.2 Compilation de IMAGEIN

Cette étape est la plus simple car la bibliothèque n'utilise pas `Qt`. Un `Makefile` est fourni pour compiler la bibliothèque (sous windows, `make` ne sera disponible qu'avec le shell `msys`). Il suffit donc de se placer dans le répertoire des sources d'IMAGEIN et de taper `make`. La bibliothèque sera compilée sous forme d'un fichier `libimagein.a`

A.3.3 Compilation de l'interface générique

La compilation de l'interface générique se fait de manière très similaire à la compilation de n'importe quel projet `Qt`. Cependant, l'utilisation d'IMAGEIN et de `Qwt` imposent quelques réglages supplémentaires. Pour commencer, placez vous dans le répertoire de l'interface générique.

4. Disponibles sur www.mingw.org

```
#####
# Automatically generated by qmake (2.01a) Mon May 21 16:28:11 2012
#####

TEMPLATE = lib
TARGET =
DEPENDPATH += . Services Utilities Widgets/ImageWidgets Widgets/NavBar
INCLUDEPATH += . Utilities Services Widgets/ImageWidgets Widgets/NavBar
INCLUDEPATH += ../ImageIn/
LIBS += -L../ImageIn -limagein -ljpeg -lpng
CONFIG += qwt static

# Input
HEADERS += Exceptions.h \
          GenericInterface.h \
# La suite n'a pas besoin d'etre modifiee...
```

FIGURE A.2 – Début du fichier GenericInterface.pro après modification

1. tapez `qmake -project`. Qmake est un utilitaire fourni par Qt qui permet de générer automatiquement un makefile. Cette première étape génère un fichier GenericInterface.pro qu'il va falloir modifier pour prendre en compte nos paramètres.
2. Modifiez le fichier GenericInterface.pro. Il faut ajouter plusieurs choses :
 - Préciser l'emplacement des fichiers .h de IMAGEIN (via la variable INCLUDEPATH)
 - Préciser qu'on veut lier la bibliothèque ImageIn (via la variable LIBS)
 - Préciser qu'on utilise la bibliothèque Qwt (via la variable CONFIG)
 - Préciser qu'on veut générer une bibliothèque statique et non une application.

Un exemple de fichier GenericInterface.pro est donné dans la figure A.2. La deuxième ligne INCLUDEPATH a été ajoutée, ainsi que les lignes LIBS et CONFIG. La ligne TEMPLATE a été modifiée de "app" à "lib".
3. tapez `qmake`. Cette commande lit le fichier GenericInterface.pro et génère un Makefile.
4. tapez `make` pour compiler l'interface générique. Le fichier de sortie s'appelle libGenericInterface.a

Selon votre installation de Qwt, il se peut qu'ajouter "qwt" à la variable CONFIG ne fonctionne pas. Dans ce cas, il faudra ajouter manuellement le chemin des fichiers .h de Qwt à la variable INCLUDEPATH, et linker la bibliothèque Qwt via la variable LIBS. Plus d'informations peuvent être trouvées sur le site de Qt ou de Qwt.

```
#####
# Automatically generated by qmake (2.01a) Sat May 26 14:35:57 2012
#####

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
INCLUDEPATH += ../ImageIn ../GenericInterface
LIBS += -L../GenericInterface -lGenericInterface -L../ImageIn -limagein -ljpeg -lpng
CONFIG += qwt

# Input
# La suite n'a pas besoin d'etre modifiee...
```

FIGURE A.3 – Début du fichier GenericInterface.pro après modification

A.3.4 Compiler l'application unitaire

Cette étape fonctionne de la même manière que la précédente : Générer un fichier .pro avec `qmake -project` ; Modifier ce fichier pour ajouter les paramètres nécessaires ; générer un makefile avec `qmake`, compiler l'application en utilisant `make`.

Un exemple de fichier .pro pour une application est donné en figure A.3. Les variables à modifier sont les mêmes, à l'exception de `TEMPLATE` (qui reste "app") et de `CONFIG`, à laquelle on n'ajoute que "qwt". Il faut également ajouter les chemins vers l'interface générique aux variables `INCLUDEPATH` et `LIBS`.

L'exécutable généré fonctionne parfaitement !

