Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

N° de laboratoire	Laboratoire 3
Étudiants	Daniel Lanthier, Charles-Antoine Barrière, Maxim Luchianciuc, Frédéric Bélanger
Codes permanents	LAND18099908, BARC03119501 LUCM290898091, BELF31018006
Cours	LOG121
Session	H2021
Groupe	04
Professeur	**Vincent Lacasse**
Chargés de laboratoire	**Simon Pichette**
Date de remise	9 avril 2021

1 — Introduction

De nos jours, les patrons de conception sont de plus en plus employés dans le design des logiciels. Ce sont des solutions éprouvées avec le temps par des experts, qui aident à la résolution de problèmes complexes. Ce projet a pour but d'utiliser un grand nombre de ces patrons afin de développer un logiciel qui permet de modifier une perspective d'image par l'entremise d'interactions diverses avec l'interface.

Notre application doit supporter les commandes nécessaires pour offrir plusieurs fonctionnalités. Parmi celles-ci, on retrouve les sauvegardes et chargements, les changements d'échelle, les translations, les copier-coller ainsi que la possibilité de défaire et refaire des altérations.

Dans ce rapport, nous présentons les détails de notre conception. Nous commençons par les choix et responsabilités de chaque classe ainsi que les patrons clés auxquels nous avons eu recours pour le développement de ce logiciel. Nous introduisons ensuite quelques diagrammes *UML* pour bien illustrer notre conception. Finalement, nous discutons des faiblesses de notre design et complétons avec une décision qui a eu un impact important sur le déroulement de notre projet.

2 — Conception

2.1 Choix et responsabilités des classes

Le tableau 2.1 résume les principales classes du logiciel. Chacune d'elle est accompagnée d'une description de ses responsabilités ainsi que d'une liste des classes dont elle dépend. Les mots qui débutent par une majuscule et sont en italique indiquent des noms de classes. Pour cette raison, ils sont invariables dans le texte.

Classe	*Responsabilités*	*Dépendances*
Application	• Contenir le « main » de l'application • Instancier MainWindow • Instancier Perspective • Donner des références vers Perspective à MainWindow et à la classe statique CommandFactory	*MainWindow* *Perspective* *CommandFactory*
MainWindow	• Être la fenêtre principale de l'application (le JFrame). • Contenir les vues PerspectivePanel et ImagePanel • Contenir les éléments d'interaction avec l'utilisateur (les boutons) • Instancier les Action et les associer aux éléments de l'interface appropriés (boutons, panneaux, etc.) • Définir les raccourcis clavier pour les actions qui les utilisent	*Perspective* *PerspectivePanel* *ImagePanel* *ZoomAction* *TranslateAction* *SaveAction* *LoadAction* *CopyAction* *PasteAction* *UndoAction* *RedoAction*
ImagePanel	• Observer Perspective pour s'actualiser lorsqu'une nouvelle image est chargée • Dessiner la version non modifiable de l'image	*Perspective*
PerspectivePanel	• Afficher la version modifiable de l'image • Observer Perspective pour s'actualiser lorsque l'image est modifiée	*Perspective* *ImageProperties*
Perspective	• Contenir l'image (BufferedImage) et ses ImageProperties • Permettre les modifications sur les ImagesProperties • Permettre de charger une nouvelle image • Aviser ses observateurs lors de changements • Faciliter la sérialisation de ses propres données pour les enregistrement et chargement d'images modifiées	*ImageProperties*

Classe	*Responsabilités*	*Dépendances*
ImageProperties	• Contenir les propriétés de l'image et permettre de les modifier	
CopyBuffer *« Singleton »*	• Contenir une copie de *ImageProperties* et en permettre la récupération	*ImageProperties*
ImageSerializer	• Gérer l'enregistrement (ou la sérialisation) d'une *Perspective* dans un fichier • Gérer la récupération d'une *Perspective* à partir d'un fichier préalablement enregistré	*Perspective*
Action *« abstract »*	• Représenter une ou plusieurs actions qu'un utilisateur peut effectuer qui mèneront à l'exécution d'une *Command* • Aviser le *CommandManager* qu'une *Action* a été faite afin qu'il puisse créer et exécuter la *Command* correspondante • Contenir et fournir son propre nom	*AvailableCommands* *CommandManager*
CopyAction	• Étendre la classe *Action* • Demander sa propre exécution au *CommandManager* lorsque l'usager utilise le raccourci clavier « [CTRL]+C »	
LoadAction	• Étendre la classe *Action* • Permettre à l'usager de sélectionner un fichier à ouvrir lorsqu'il appuie sur le bouton « *Load* » • Demander sa propre exécution au *CommandManager* lorsque l'usager a complété la sélection du fichier	
PasteAction	• Étendre la classe *Action* • Permettre à l'usager de sélectionner les propriétés de l'image qu'il veut « coller » lorsqu'il utilise le raccourci clavier « [CTRL]+V » • Demander sa propre exécution au *CommandManager* lorsque l'usager a fait son choix	
RedoAction	• Étendre la classe *Action* • Demander au *CommandManager* sa propre exécution lorsque l'usager utilise le raccourci clavier « [CTRL]+Y » ou s'il appuie sur le bouton « *Redo* »	
SaveAction	• Étendre la classe *Action* • Permettre à l'usager de sélectionner un fichier pour l'enregistrement de la *Perspective* lorsqu'il utilise le raccourci clavier « [CTRL]+S » ou s'il appuie sur le bouton « *Save* »	

Classe	*Responsabilités*	*Dépendances*
	• Demander sa propre exécution au *CommandManager* lorsque l'usager a complété la sélection du fichier	
TranslateAction	• Étendre la classe *Action* • Enregistrer les mouvements du pointeur lorsque l'utilisateur clique sur *PerspectivePanel* et déplace la souris en maintenant le bouton enfoncé • Demander sa propre exécution au *CommandManager* lorsque l'utilisateur a relâché le bouton de la souris	
UndoAction	• Étendre la classe *Action* • Demander au *CommandManager* sa propre exécution lorsque l'usager utilise le raccourci clavier « [CTRL]+Z » ou s'il appuie sur le bouton « *Undo* »	
ZoomAction	• Étendre la classe *Action* • Enregistrer les mouvements de la roulette de la souris lorsque l'utilisateur l'actionne et que le pointeur est au-dessus de *PerspectivePanel* • Demander sa propre exécution au *CommandManager*	
Command *« Interface »*	• Représenter une commande avec les fonctions à exécuter	
BaseCommand *« abstract »*	• Implémenter l'interface *Command* • Contenir une référence vers *Perspective* pour que les différentes *BaseCommand* puissent y faire les modifications appropriées	*Command* *Perspective*
UndoableCommand *« abstract »*	• Étendre la classe abstraite *BaseCommand* • Ajouter les méthodes « *undo* » et « *redo* » aux *Command* qui sont annulables	*BaseCommand*
ZoomCommand	• Étendre *UndoableCommand* • Modifier le zoom de l'image qui est dans *Perspective* • S'ajouter à la pile des commandes annulables de *CommandManager* une fois exécutée • Gérer l'annulation de ses propres modifications (le « *redo* ») et s'ajouter à la pile des commandes « refaisables » (*redoStack*) de *CommandManager*	*UndoableCommand* *CommandManager*
TranslateCommand	• Étendre *UndoableCommand* • Modifier la position de l'image qui est dans *Perspective* • S'ajouter à la pile des commandes annulables de *CommandManager* une fois exécutée	*UndoableCommand* *CommandManager*

Classe	*Responsabilités*	*Dépendances*
	• Gérer l'annulation de ses propres modifications (le « *redo* ») et s'ajouter à la pile des commandes « refaisables » (*redoStack*) de *CommandManager*	
SaveCommand	• Étendre *BaseCommand* • Sauvegarder l'image qui est dans *Perspective* et ses *ImageProperties* dans un fichier	*BaseCommand* *ImageSerializer*
LoadCommand	• Étendre *BaseCommand* • Récupérer une image à partir d'un fichier et la mettre dans *Perspective* (si le chargement se fait à partir d'un fichier sérialisé créé par *SaveCommand*, restaurer l'image et ses *ImageProperties*)	*BaseCommand* *ImageSerializer*
CopyCommand	• Étendre *BaseCommand* • Copier les *ImageProperties* de *Perspective* et les mettre dans *CopyBuffer*	*BaseCommand* *CopyBuffer*
PasteCommand	• Étendre *UndoableCommand* • Changer certaines propriétés (au choix de l'utilisateur) de *Perspective* à partir des *ImageProperties* copiées dans *CopyBuffer* • S'ajouter à la pile des commandes annulables de *CommandManager* une fois exécutée • Gérer l'annulation de ses propres modifications (le « *redo* ») et s'ajouter à la pile des commandes « refaisables » (*redoStack*) de *CommandManager*	*UndoableCommand* *CommandManager* *CopyBuffer*
CopyBuffer *« Singleton »*	• Contenir une copie de *ImageProperties* et en permettre la récupération	*ImageProperties*
UndoCommand	• Étendre *BaseCommand* • Demander au *CommandManager* de défaire la dernière *Command* annulable effectuée	*BaseCommand* *CommandManager*
RedoCommand	• Étendre *BaseCommand* • Demander au *CommandManager* de refaire la dernière *Command* annulée	*BaseCommand* *CommandManager*
CommandFactory *« Static »*	• Créer des *Command* à partir des *Action* correspondantes • Contenir une référence vers la *Perspective* et la passer aux *Command* lorsqu'elles sont créées	*AvailableCommands* *Perspective* *ZoomAction* *TranslateAction* *SaveAction* *LoadAction* *CopyAction* *PasteAction* *UndoAction* *RedoAction* *CopyCommand* *LoadCommand*

Classe	*Responsabilités*	*Dépendances*
		PasteCommand *RedoCommand* *UndoCommand* *SaveCommand* *TranslateCommand* *ZoomCommand*
CommandManager *« Singleton »*	• Demander à *CommandFactory* de créer les *Command* à partir des *Action* • Demander aux *Command* de s'exécuter • Contenir et gérer les piles de *Command* annulables (*undoStack*) et « refaisables » (*redoStack*) • Lors des *undo*/*redo*, prendre la *Command* qui est au-dessus de la pile appropriée et lui demander de se défaire/refaire	*MainWindow* *Observer* *Command* *CommandFactory*

Tableau 2.1 — choix et responsabilités des classes

2.2 Diagrammes de classe (*UML*)

Cette section présente deux diagrammes (*UML)* qui permettent de voir rapidement les différentes classes de l'application ainsi que les liens qui les unissent. Le diagramme à la figure 2.1 montre l'architecture générale du programme alors que celui de la figure 2.2 met l'accent sur la gestion des commandes. Nous avons également inclus une version complète pour l'ensemble du logiciel, à l'Annexe 1.

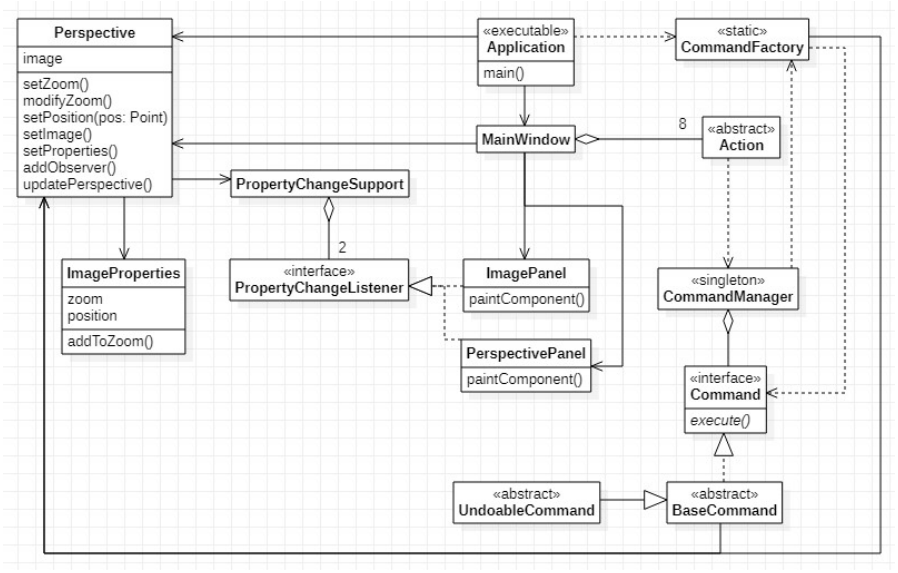

Figure 2.1 — diagramme de classe (*UML)* de l'architecture du logiciel

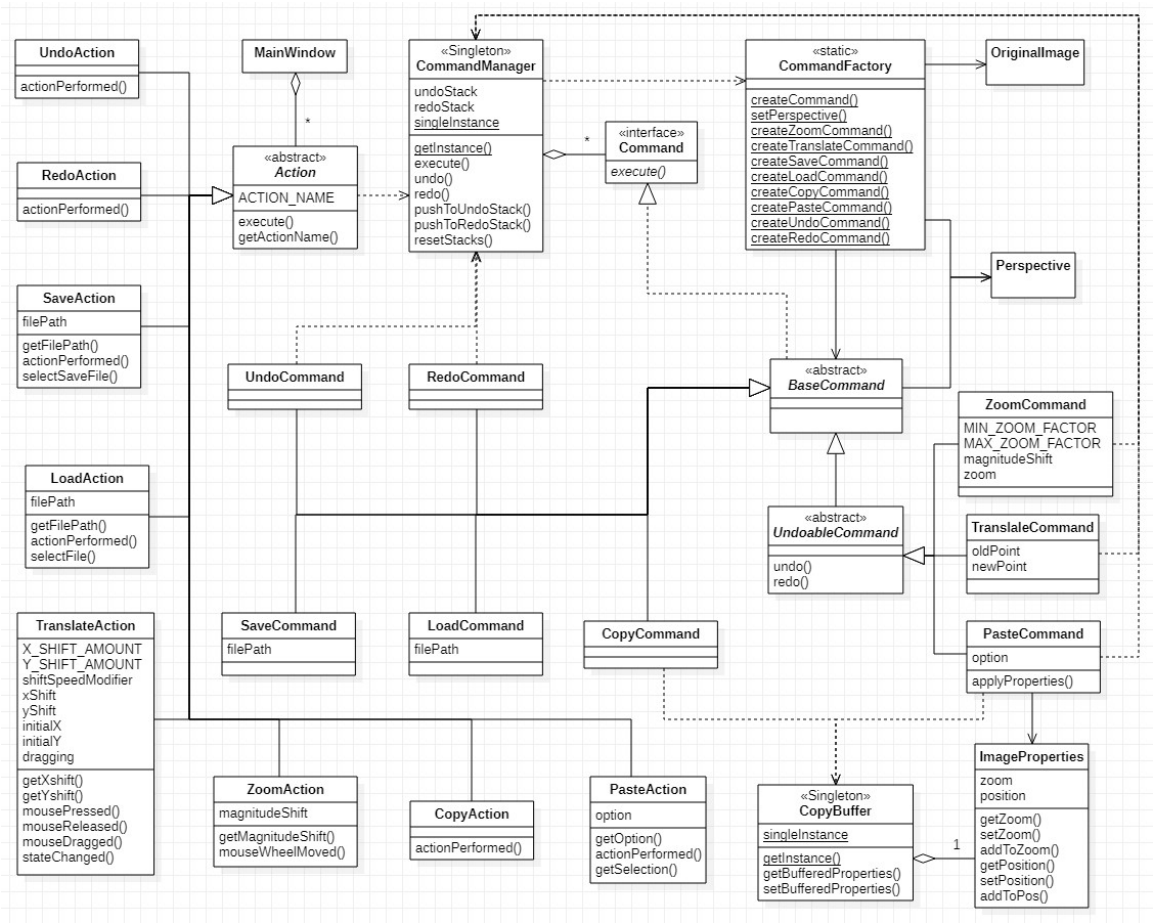

Figure 2.2 — diagramme de classe (*UML)* qui met l'emphase sur la gestion des commandes

2.3 Utilisation des patrons de conception

2.3.1 Patron *Observer*

Dans notre application, on utilise le patron *Observer* pour que *PerspectivePanel* et *ImagePanel* soient avertis des changements ayant lieu dans la classe *Perspective*. Lorsqu'une modification est détectée dans l'instance de *Perspective*, on redessine *PerspectivePanel*. Si une nouvelle image est chargée à l'aide de la commande « *Load* », *ImagePanel* est également mis à jour.

Patron *Observer*	Nom dans le laboratoire
Subject	*PropertyChangeSupport*
ConcreteSubject	*Perspective*
attach(o : *Observer*)	*addObserver*(*listener*: *PropertyChangeListener*)
dettach(o: *Observer*)	----
Observer	*PerspectivePanel* *ImagePanel*

Tableau 2.2 — correspondances entre les noms originaux des classes du patron *Observer* et ceux du laboratoire

2.3.2 Patron *Command*

Le but du patron *Command* est d'encapsuler une fonctionnalité dans un objet qui contient tout ce dont il a besoin pour s'exécuter. Dans notre logiciel, nous l'employons pour gérer les différentes commandes offertes. Avec ce patron, on peut implémenter facilement les *undo*/*redo*, puisque les fonctionnalités encapsulées peuvent être conservées dans des piles et sont aisément manipulables.

Patron *Command*	Nom dans le laboratoire
Command	*Command*
<<abstract>> ConcreteCommand	*BaseCommand*
	UndoableCommand
ConcreteCommand	*SaveCommand*
	LoadCommand
	TranslateCommand
	ZoomCommand
	CopyCommand
	PasteCommand
	UndoCommand
	RedoCommand
Client	*CommandManager*

Tableau 2.3 — correspondances entre les noms originaux des classes du patron *Command* et ceux du laboratoire

2.3.3 Patron *MVC*

Le patron *MVC* est employé pour créer une séparation entre trois éléments : l'information concernant l'image, l'affichage à l'écran ainsi que les interactions de l'utilisateur avec le logiciel. On augmente ainsi la modularité du logiciel puisqu'on peut effectuer des changements à l'une des trois entités sans affecter significativement les deux autres.

- *MainWindow*, *ImagePanel* et *PerspectivePanel* s'occupent d'afficher l'image à l'écran ;
- *CommandManager*, *Command* et *Action* contrôlent les interactions entre l'utilisateur et le logiciel ;
- *Perspective* contient l'information qui doit être affichée et les méthodes pour l'altérer.

Patron *MVC*	Nom dans le laboratoire
View	*MainWindow*
	ImagePanel
	PerspectivePanel
Controler	*CommandManager*
	Command
	Action
Model	*Perspective*

Tableau 2.4 — correspondances entre les noms originaux des classes du patron *MVC* et ceux du laboratoire

2.3.4 Patron *Singleton*

Nous utilisons ce patron afin d'avoir une seule instance de *CommandManager* et que celle-ci soit facilement accessible. On s'assure ainsi que les piles de commandes *undo* et *redo* soient uniques pour l'ensemble du logiciel.

Nous employons également ce patron pour le *CopyBuffer.* Celui-ci doit garder uniquement la dernière copie effectuée et la remplacer par toute nouvelle copie faite, peu importe la situation.

Patron *Singleton*	Nom dans le laboratoire
Singleton	*CommandManager*
Singleton	*CopyBuffer*

Tableau 2.5 — correspondances entre les noms originaux des classes du patron *Singleton* et ceux du laboratoire

2.3.5 Patron Factory Method

Le patron *Factory Method* permet d'instancier des classes en fonction des paramètres qui sont passés à la méthode créatrice (*factoryMethod* ou *createCommand*).

Dans l'application, ce patron sert à créer tous les types de *Command* à partir des *Action* correspondantes. On s'assure ainsi d'une meilleure encapsulation des commandes, car *CommandFactory* est la seule entité à en connaitre la nature exacte. La *CommandFactory* possède également une référence vers la *Perspective*, qu'elle peut passer à chaque *Command* lorsqu'elles sont créées.

Patron Factory Method	***Nom dans le laboratoire***
Creator	*----*
Product	*BaseCommand*
ConcreteCreator	*CommandFactory*
ConcreteProduct	*SaveCommand*
	LoadCommand
	TranslateCommand
	ZoomCommand
	CopyCommand
	PasteCommand
	UndoCommand
	RedoCommand
factoryMethod	*createCommand*

Tableau 2.6 — correspondances entre les noms originaux des classes du patron *Factory Method* et ceux du laboratoire

2.4 Faiblesse de la conception

Notre conception comporte trois faiblesses principales. Premièrement, plusieurs commandes ont accès à une instance de *Perspective* sans interagir avec celle-ci. Nous aurions pu créer une autre sous-classe de *BaseCommand* qui aurait contenu une référence vers *Perspective*. On aurait alors choisi uniquement les classes qui ont besoin de cette référence pour en hériter. Nous aurions ainsi évité des couplages qui n'ont pas lieu d'être.

Deuxièmement, lors de la création de *MainWindow*, nous passons une instance de *Perspective* qui est ensuite relayée à un *ImagePanel* et un *PerspectivePanel*. Cette approche crée un couplage

inutile entre la classe *Perspective* et la classe *MainWindow*. Cela aurait pu être évité, puisque la classe *MainWindow* n'utilise pas Perspective. Une meilleure approche aurait été de créer des mutateurs dans les classes *ImagePanel* et *PerspectivePanel,* pour leur donner directement la référence requise.

Finalement, l'application génère plusieurs petites *Action* à faire dès que l'usager interagit avec le programme, au lieu d'attendre que l'action soit complétée en entier. Par exemple, lorsque l'utilisateur actionne la roulette de la souris, au lieu d'avoir une seule *Action* qui couvre l'ensemble du mouvement, celui-ci est divisé en « fragments » de mouvement. Si le zoom total pour le mouvement de la roulette est de 20 %, on pourrait avoir cinq *Action* qui font chacune 4 % de mise à l'échelle.

Cela est problématique, car lors de l'annulation de la *Command* correspondante, l'utilisateur doit faire cinq « *undo* » alors qu'il a l'impression de n'avoir effectué qu'une seule manipulation. Une meilleure approche aurait pu être d'utiliser le patron *Memento* pour prendre une image de l'état de la *Perspective* avant le début de la « série d'Action » pour la restaurer lors d'un « *undo* ».

2.5 Diagrammes de séquence (*UML*)

2.5.1 Dynamique de l'architecture *MVC* : effectuer une commande zoom

Le diagramme de la figure 2.3 illustre ce qui se produit lorsque l'utilisateur actionne la roulette de la souris pour faire un zoom. Tout d'abord, *ZoomAction* récupère et emmagasine l'amplitude du mouvement dans le *MouseWheelEvent* actionné. Elle demande ensuite sa propre réalisation au *CommandManager*. Celui-ci prend la *ZoomAction* et utilise *CommandFactory* afin de créer la *ZoomCommand* correspondante. Il exécute ensuite cette commande.

Après avoir vérifié si le changement d'échelle demandé ne fera pas dépasser le zoom actuel de ses valeurs minimale et maximale, *ZoomCommand* va appeler le *setZoom*() de *Perspective* qui va effectuer l'ajustement dans ses *ImageProperties*. Une fois le tout complété, *Perspective* avise son observateur *PerspectivePanel*. Celui-ci récupère l'image et ses propriétés afin d'actualiser l'affichage. Pour terminer, la *ZoomCommand* va demander au *CommandManager* de l'ajouter à la pile de commandes « *undo* » afin que l'utilisateur puisse éventuellement l'annuler.

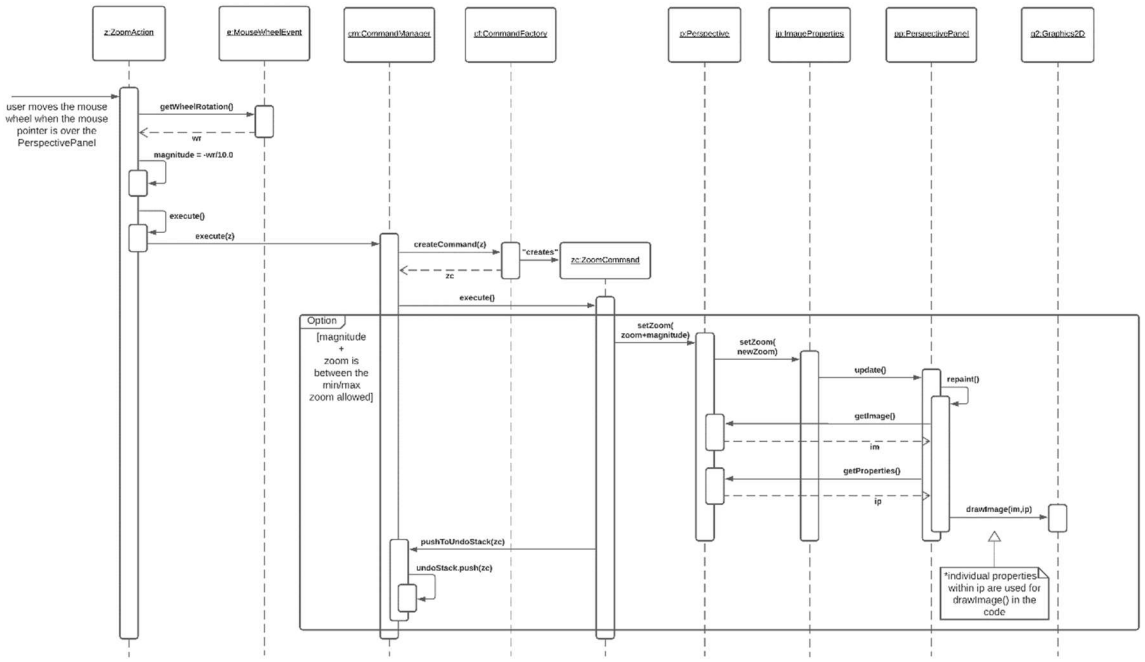

Figure 2.3 — diagramme de séquence (*UML)* qui illustre la dynamique de l'architecture *MVC* et du patron *Command* lorsque l'utilisateur veut faire un zoom

2.5.2 Dynamique de l'architecture *MVC* : défaire une commande zoom

Le diagramme de la figure 2.4 montre le fonctionnement de notre implémentation du patron *Command* et de l'architecture *MVC*, lorsque l'annulation d'une *ZoomCommand* est demandée. Tout comme à la section 2.5.1, lorsqu'on enclenche le bouton « *undo* », on génère une *UndoAction* qui sera éventuellement transformée en *UndoCommand* par le *CommandManager* et la *CommandFactory*. Lors de son exécution, *UndoCommand* utilise la méthode « *undo*() » du *CommandManager*. Celui-ci retire la *ZoomCommand* qui est sur le dessus de la pile correspondante et appelle la méthode « undo() » de la commande.

ZoomCommand va alors se servir de la méthode « *modifyZoom*() » de *Perspective,* mais en donnant en paramètre l'inverse de son amplitude (*magnitudeShift*) originel. Si celui-ci était positif, il sera négatif (et vice versa). Une fois la manipulation complétée, *Perspective* avise son observateur *PerspectivePanel*. Celui-ci récupère l'image et ses propriétés afin d'actualiser l'affichage. Pour terminer, *ZoomCommand* demande au *CommandManager* d'être ajoutée à la pile « *redo* » afin que l'utilisateur puisse éventuellement la refaire.

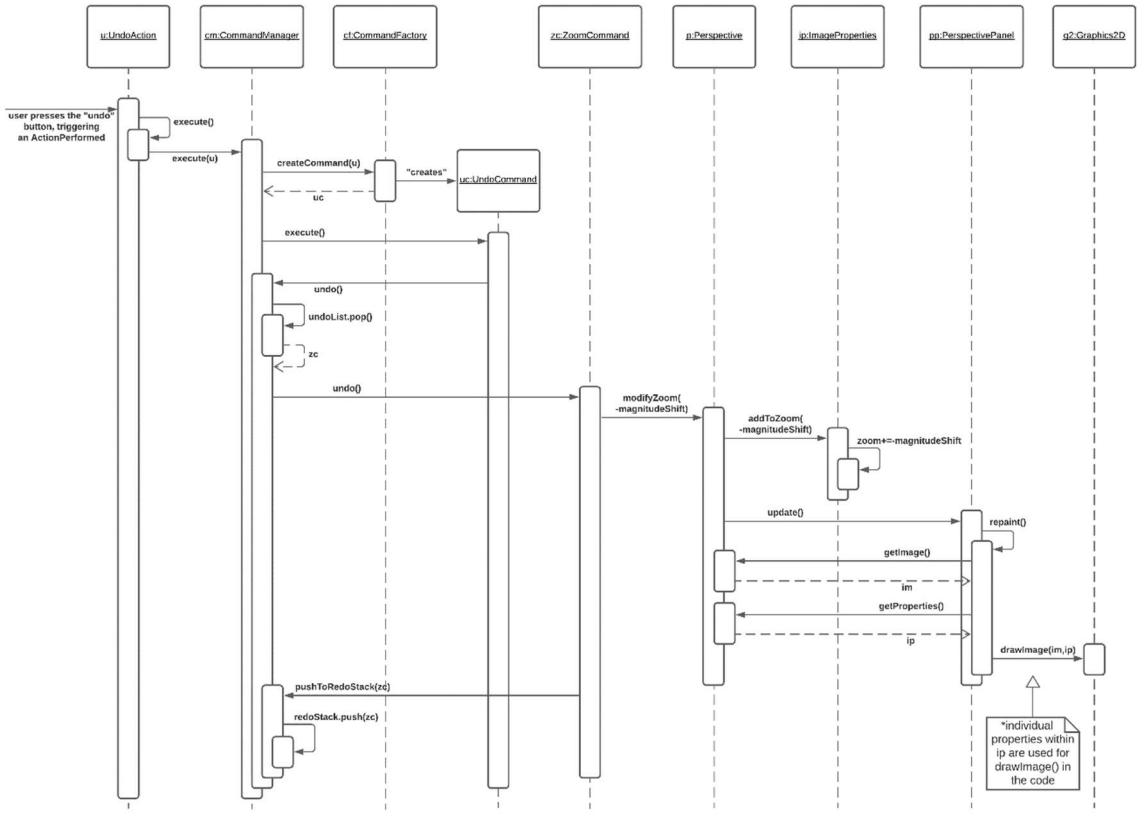

Figure 2.4 — diagramme de séquence (*UML)* qui illustre la dynamique de l'architecture MVC et du patron Command lorsque l'utilisateur annule une commande zoom

3 — Décisions de conception/d'implémentation

3.1 Décision 1 : Conception des commandes du logiciel

3.1.1 Contexte

Dans notre logiciel, ce ne sont pas toutes les commandes qui peuvent être défaites et refaites (*undo/redo*). Par exemple, on peut annuler une translation de l'image, mais pas une sauvegarde. Nous devions décider comment nous implanterions cette particularité.

3.1.2 Solution 1

Avec le patron *Command*, toutes les commandes doivent implémenter une interface commune qui comporte la fonction « *execute*() ». On pourrait ajouter les méthodes « *undo*() » et « *redo*() » à cette interface. Pour déterminer quelles commandes sont annulables ou pas, chacune d'elles aurait une variable booléenne. Lors de l'exécution, le *CommandManager* serait ainsi en mesure d'identifier celles qui sont annulables pour les ajouter à sa pile de commandes « *undo* ».

L'avantage de cette solution est qu'elle permet d'utiliser un seul gestionnaire pour toutes les commandes du système. L'implémentation serait plus rapide à programmer. De plus, une gestion centralisée aiderait à la maintenabilité et à l'extensibilité du logiciel puisqu'on serait plus aisément en mesure de suivre son fonctionnement.

Cette solution comporte cependant trois désavantages majeurs. Premièrement, toutes les commandes devront définir les méthodes « *undo*() » et « *redo*() », même celles qui ne s'en servent pas. On crée du code qui est superflu et qui peut porter à confusion. Pourquoi une commande qui n'est pas annulable aurait-elle une méthode qui permet de l'annuler ?

Deuxièmement, l'un des objectifs du patron *Command* est d'encapsuler les commandes dans des objets. Chacun de ceux-ci doit contenir tout ce dont il a besoin pour réaliser la fonctionnalité en entier. Avec cette solution, on relègue une partie de l'exécution au *CommandManager.* Celui-ci doit accéder à chaque commande pour déterminer si elle doit être ajoutée à la pile « *undo* » ou pas.

Troisièmement, nous avons déterminé que la création des commandes se ferait à l'aide du patron *Factory Method*. Avec ce patron, seule la classe qui fabrique les objets doit en connaitre la nature exacte. On brise ce principe lorsque le *CommandManager* vérifie si les commandes sont annulables.

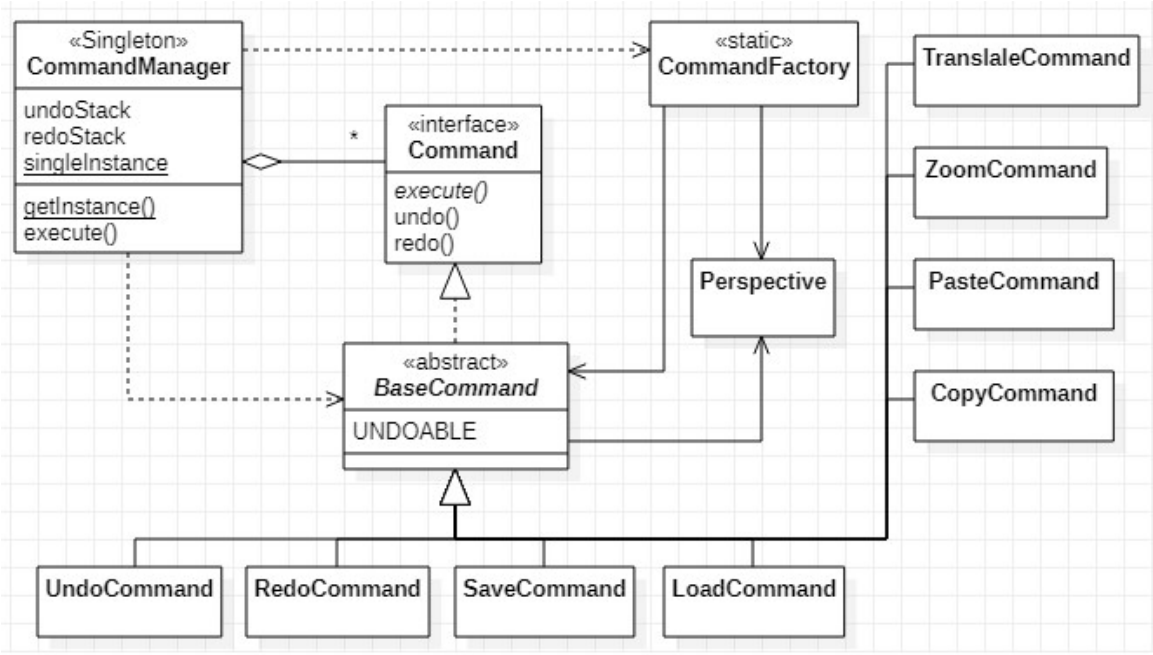

Figure 3.1 — diagramme de classe de la première solution envisagée

3.1.3 Solution 2

On pourrait créer deux gestionnaires de commande. L'un s'occuperait de celles qui sont annulables et l'autre de celles qui ne le sont pas. Dans notre design, ce sont les actions qui demandent au gestionnaire leurs propres exécutions. Elles pourraient donc être associées à celui qui convient, selon leurs annulabilités.

Avec cette approche, on évite que les commandes aient des méthodes non essentielles à leurs fonctions. On respecte également mieux le principe du patron *Factory Method* puisque la *CommandFactory* est la seule à connaitre la nature des objets qu'elle fabrique.

On complexifie cependant l'implémentation. Deux gestionnaires doivent être créés et le nombre de classes est augmenté. De plus, la maintenance et les possibles extensions du logiciel pourraient être plus complexes. Quelqu'un qui n'est pas familier avec notre design pourrait avoir plus de difficulté à naviguer ou à modifier un système avec deux gestionnaires distincts.

On pourrait aussi se demander ce qu'on ferait si une commande qui nécessite un autre type de traitement particulier était à implémenter. Devrait-on créer un nouveau gestionnaire pour chaque commande qui se démarque des autres ?

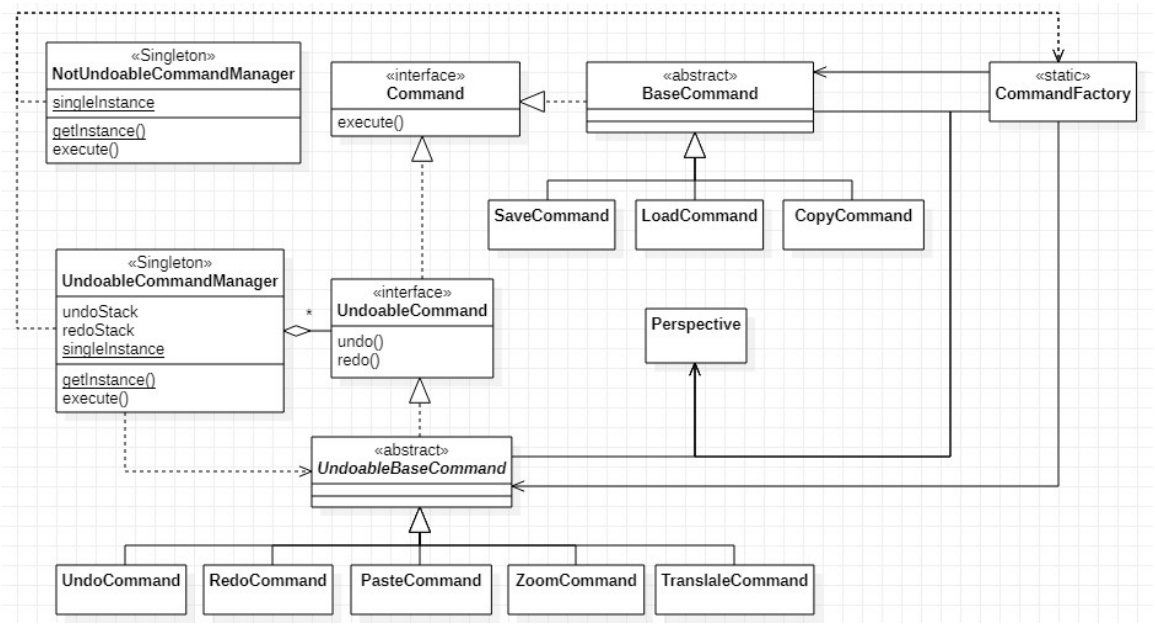

Figure 3.2 — diagramme de classe de la deuxième solution envisagée

3.1.4 Choix et justification — un compromis

Nous avions initialement opté pour la première solution. Vu l'envergure limitée du projet, il nous semblait exagéré de créer deux gestionnaires distincts. Lors de l'implémentation, nous avons décidé de modifier notre conception pour adopter un compromis entre les deux options préalablement envisagées.

Comme vu à la section 2.2, nous avons conservé un système à gestionnaire unique. Mais au lieu d'utiliser une variable booléenne pour identifier les commandes annulables, nous les avons fait hériter d'une nouvelle classe abstraite nommée *UndoableCommand*. Celle-ci déclare la méthode « *undo*() » et définit la méthode « *redo*() ». De plus, nous avons fait en sorte que toutes les commandes annulables demandent elles-mêmes au gestionnaire d'être ajouté aux piles « *undo* » et « redo ».

Tout cela fait en sorte que notre implémentation finale respecte mieux les principes des patrons *Factory Method* et *Command*. *CommandFactory* est la seule à connaitre le contenu des commandes et chacune d'elles s'occupe de sa propre exécution du début à la fin. La maintenabilité et l'extensibilité du logiciel sont également améliorées. On peut aisément ajouter des classes qui étendent celles qui sont existantes pour implémenter de nouveaux types de commandes.

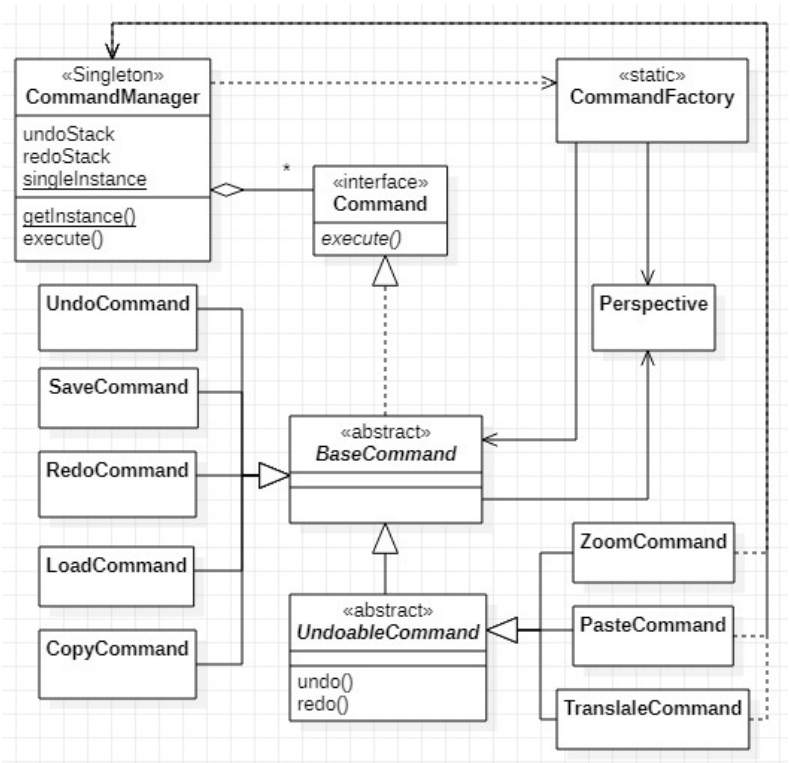

Figure 3.3 — diagramme de classe de la solution adoptée

4 — Conclusion

Pour conclure, l'objectif de ce laboratoire était de développer une application qui permet à un utilisateur de modifier une image. Notre solution répond à tous les critères demandés pour ce logiciel. Elle permet de zoomer, d'effectuer des translations, de copier et coller les propriétés de l'image ainsi que de défaire et refaire des commandes préalables. De plus, l'utilisateur peut charger et sauvegarder une version modifiée de l'image (la *Perspective*).

À l'aide des patrons *Command*, *Observer*, *Factory Method*, *MVC* et *Singleton*, nous avons développé un logiciel hautement modulable et réutilisable qui respecte bien les principes de la programmation orientée objet. Notre interface graphique permet également à l'utilisateur un excellent contrôle sur les modifications qu'il peut apporter.

Malgré tout cela, notre conception n'est pas parfaite. Certaines classes, comme *MainWindow* et *PerspectivePanel* ou encore *MainWindow* et *ImagePanel*, sont inutilement couplées. Il serait intéressant d'envisager une seconde phase pour corriger ces défauts.

Comme ce logiciel permet déjà la modification d'images, il serait également pertinent, lors d'un développement futur, d'implémenter de nouvelles fonctionnalités pour le transformer en application de dessin libre. Par exemple, on pourrait offrir la possibilité de rogner l'image ou d'en améliorer le contraste. Si on combine ces ajouts à une optimisation de l'interface, nous aurions une solide base pour un logiciel qui offre de multiples possibilités.

Annexe 1 — diagramme de classe UMC de l'ensemble du logiciel