

**Département de génie logiciel et des TI**

Rapport de laboratoire

|  |  |
| --- | --- |
| **No de laboratoire** | Laboratoire 3 |
| **Étudiant(s)** | Charles Fleury, Thomas Lavergne |
| **Code(s) permanent(s)** | FLEC06129801, LAVT80050007 |
| **Cours** | LOG121 |
| **Session** | HIVER-20 |
| **Groupe** | 03 |
| **Professeur** | **Cedric St-Onge** |
| **Chargés de laboratoire** | **Hind Errahmouni** |
| **Date de remise** | 1er avril 2020 |

# 1 Introduction

Après avoir appris les différents patrons de conception vu en cours, nous avons pu les appliquer dans les précédents laboratoires pour la réalisation de différentes applications : patrons Observateur et Stratégie pour une chaîne de production d’avions, ou encore les patrons Stratégie, Méthode Template et Itérateur pour un cadriciel de jeu de dés.

Dans ce laboratoire, nous allons utiliser toutes les connaissances acquises en classes pour réaliser une application de modification de perspective d’image : certains patrons seront obligatoires (notamment Command, Observer et Singleton), mais l’utilisation de patrons supplémentaires est libre, voire conseillée, tout cela dans le but de faire une application répondant aux « besoins » de la programmation orientée objet, en utilisant une conception qui « facilite l’implémentation et la maintenance du logiciel » (Extrait des notes de cours, *Révision rapide et discussion des principes de base de la conception orientée objet*).

L’application que nous avons du réaliser était, comme énoncé précédemment, une application de modification de perspective pour une image, en modifiant celle-ci avec des commandes permettant de réaliser une translation ou un zoom sur l’image. En plus de ça, nous devions pouvoir ouvrir une image et l’afficher sur l’interface (obligatoire pour pouvoir effectuer des modifications dessus), pouvoir annuler ou refaire une commande (undo / redo), ou encore sauvegarder l’image modifiée pour pouvoir l’ouvrir plus tard, en appliquant les mêmes modifications effectuées.

Nous avons pu réaliser l’ensemble de ces fonctionnalités pour notre application, donc nous allons vous présenter notre conception ainsi que les décisions que nous avons prises pour ce projet.

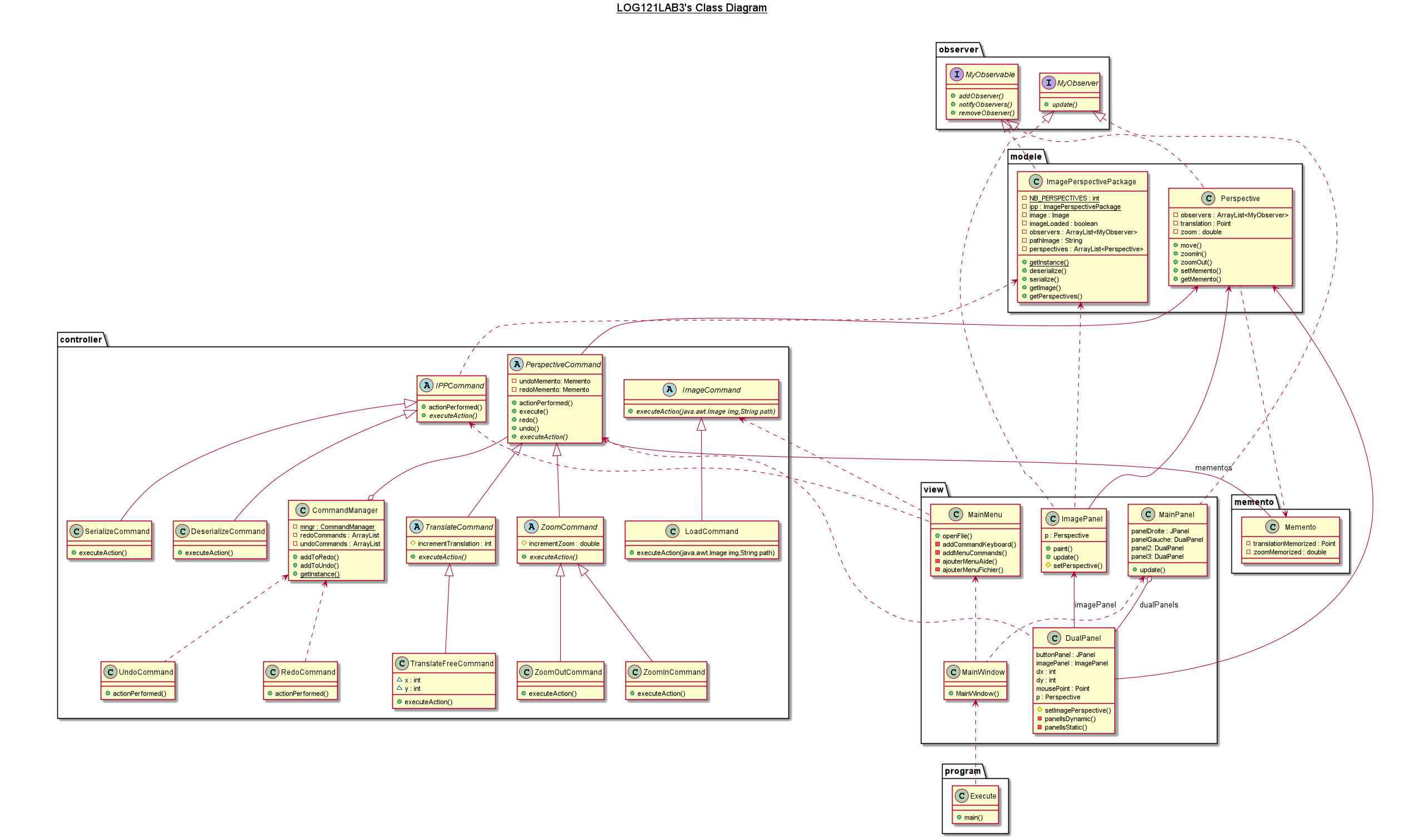
Nous allons commencer par vous présenter la conception en listant les différentes classes avec leurs responsabilités et dépendances, puis nous présenterons notre diagramme de conception ainsi que les différents patrons utilisés. Enfin, nous terminerons la partie Conception en parlant des faiblesses de celle-ci dans ce projet, ainsi que deux diagrammes de séquences de l’application. Finalement, nous finirons la présentation de ce projet en parlant de décisions d’implémentation dans la conception.

# 2 Conception

## 2.1 Choix et responsabilités des classes

|  |  |  |
| --- | --- | --- |
| *Classe* | *Responsabilités* | *Dépendances* |
| * ImagePerspectivePackage | * Classe Singleton contenant l’image, les perspectives ainsi que les méthodes de sérialisation et désérialisation | * Perspective, Serializable |
| * Perspective | * Classe gérant les perspectives de l’image : on utilise le patron Observer dessus pour pouvoir notifier lorsque la Perspective est modifiée, et ainsi pouvoir mettre à jour l’affichage sur la Vue. * Le patron Memento est utilisé pour pouvoir appliquer les commandes défaire et refaire sur la perspective | * MyObservable, Serializable, Memento |
| * IPPCommand | * Classe abstraite utilisée pour les commandes de sérialisation et désérialisation |  |
| * SerializeCommand | * Classe permettant la commande de sérialisation | * IPPCommand, ItemPackagePerspective |
| * DeserializeCommand | * Classe permettant la commande de déserialisation | * IPPCommand, ItemPackagePerspective |
| * CommandManager | * Classe Singleton permettant de gérer l’historique des commandes permettant de réaliser plus facilement les commandes défaire et refaire |  |
| * ImageCommand | * Classe abstraite impactant sur l’image |  |
| * LoadCommand | * Commande permettant de charger une image dans l’instance d’ImagePerspectivePackage | * ImagePerspectivePackage |
| * PerspectiveCommand | * Classe abstraite gérant les commandes impactant sur les perspectives | * Memento, Perspective |
| * RedoCommand | * Classe gérant la commande refaire sur les perspectives | * Perspective, PerspectiveCommand |
| * UndoCommand | * Classe gérant la commande défaire sur les perspectives | * Perspective PerspectiveCommand |
| * TranslateCommand | * Classe abstraite gérant les commandes de translation sur les perspectives | * PerspectiveCommand |
| * TranslateFreeCommand | * Classe gérant les translations de l’image sur la perspective. | * TranslateCommand, Perspective |
| * ZoomCommand | * Classe abstraite gérant les zooms et dézoom sur l’image | * PerspectiveCommand |
| * ZoomInCommand | * Classe pour la commande de zoom sur l’image | * ZoomCommand, Perspective |
| * ZoomOutCommand | * Classe pour la commande de dézoom sur l’image | * ZoomCommand, Perspective |
| * Memento | * Classe utilisée pour réaliser les commandes défaire et refaire | * Perspective |
| * MyObservable | * Interface définissant les objets observés (ici , Perspective) | * MyObserver |
| * MyObserver | * Interface définissant les objets notifiés lors d’un changement de la Perspective |  |
| * ImagePanel | * Classe Observer permettant d’afficher l’image sélectionnée ainsi que d’appliquer la Perspective dessus. Si elle est notifiée, alors elle va relancer l’affichage de l’image et de sa perspective en appliquant les dernières modifications. | * JPanel, Perspective, MyObserver |
| * DualPanel | * Classe JPanel sur lesquels les images seront affichées : * Il peut être gros (panel de gauche) et affichera ainsi une grande image et / ou dynamique (qui acceptera les modifications sur l’image). S’il est dynamique, l’image pourra être modifiée et des boutons apparaîtront au-dessus de celle-ci. * De plus, les commandes de Zoom et Translation seront effectuées depuis cette classe : le panel affiche des actions directement. | * JPanel, ImagePanel, Perspective |
| * MainMenu | * Classe JMenuBar qui permet d’ajouter les fonctions de l’application, notamment pour ouvrir une nouvelle Image, l’enregistrer avec ses Perspectives, ou encore ouvrir un fichier d’image avec Perspectives. Permet aussi d’accéder aux commandes défaire et refaire | * JMenuBar |
| * MainPanel | * Classe permettant d’initialiser toute la création et mise en place des Panels ainsi que leurs paramètres correspondants. | * JPanel, ImagePerspectivePackage, DualPanel |
| * MainWindow | * Classe permettant d’initialiser et créer l’application dans son intégralité en ajoutant MainPanel et MainMenu dans l’application. | * MainPanel, MainMenu, JFrame |

## 2.2 Diagramme des classes

­­­­­ ­­­­­­

## 2.3. Utilisation des patrons de conception

### 2.3.1 Patron Observateur



Son utilisation est appropriée, car les ImagePanels doivent se mettre à jour à chaque changement de la Perspective auquelle ils sont abonnés. La Perspective ne doit pas connaître les panels en concordance avec la bonne architecture MVC. La relation sujet-observateur entre la perspective et les panels est donc idéale. Cette relation est aussi appliquée entre le MainPanel et ImagePerspectivePackage, car lors de la deserialisation, MainPanel doit faire une mise à jour en liant les ImagePanels aux Perspectives chargées.

### 2.3.2 Patron Commande



\*\*ZoomCommand et TranslateCommand sont des classes abstraites, intermédiaires entre PerspectiveCommand et leur ConcreteCommands respectives.

Le patron commande est idéal dans notre application, car les commandes peuvent être déclenchées de diverses manières, soit par le panel directement, ou par des boutons, et cela nous permet de réduire le couplage (et la cohésion) en permettant aux vues de s’occuper seulement de l’affichage et la récupération des données du modèle.

### 2.3.3 Patron Singleton



Le patron Singleton s’applique bien car le CommandManager et ImagePerspectivePanel doivent être uniques et généralement accessibles, assurant respectivement ainsi l’unicité de l’enregistrement des commandes et des données du modèle.

### 2.3.4 Patron Memento



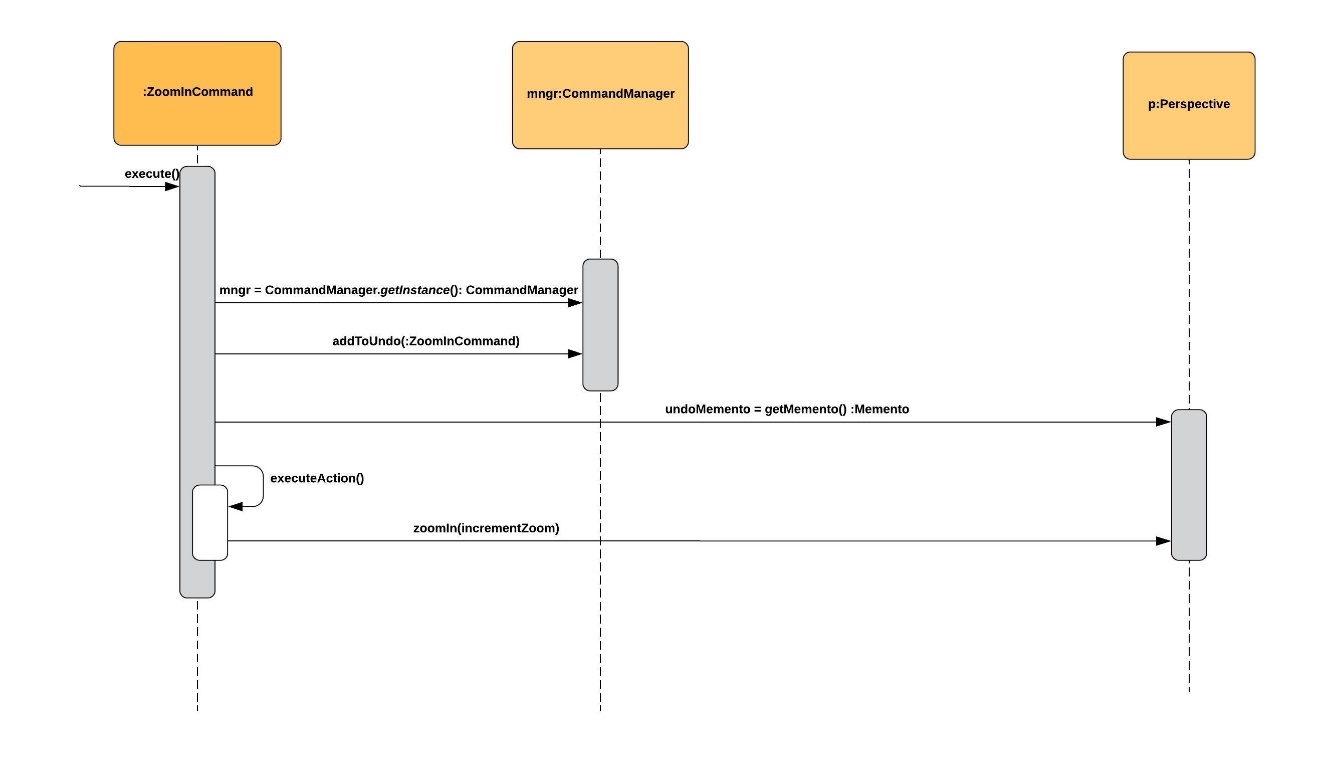
Le patron Memento est idéal pour notre application car nous avons décidé de sauvegarder des états de Perspective pour appliquer les fonctionnalités de défaire et refaire des commandes, et ce patron permet ces fonctionnalités tout en réduisant le couplage et en assurant une bonne répartition des responsabilités.

## 2.4 Faiblesses de la conception

1. Pour pouvoir couvrir le maximum de fonctionnalités demandées dans l’énoncé, nous avons utilisé le patron Command. Seulement, il est possible que nous ayons créé trop de classes pour ces fonctionnalités : en effet, nous avons effectué un changement sur les classes de translation. Au début, nous avions créé quatre classes afin de réaliser les translations de l’image : une classe pour chacun des sens possibles (haut, bas, droite et gauche). Seulement, après avoir créé la classe TranslateFreeCommand, permettant de réaliser la translation libre en utilisant le click & drag de la souris ainsi que les coordonnées de celle-ci pour réaliser la translation, nous avons réalisé que nous pouvions nous passer des quatre classes pour n’utiliser que la nouvelle pour les translations, peu importe le sens de celles-ci.
2. Nous aurions ainsi pu appliquer le même principe sur les commandes Zoom. De plus, il n’est pas nécessaire de garder les classes permettant la sérialisation des perspectives, étant donné que les méthodes sont implémentées directement dans ImagePerspectivePackage, seulement nous avons choisi de les passer sous forme de commande afin de garder l’utilisation du patron Commande sur toutes les commandes disponibles sur l’application (excepté les commandes de sortie etc).

## 2.5.Diagramme de séquence (uml)

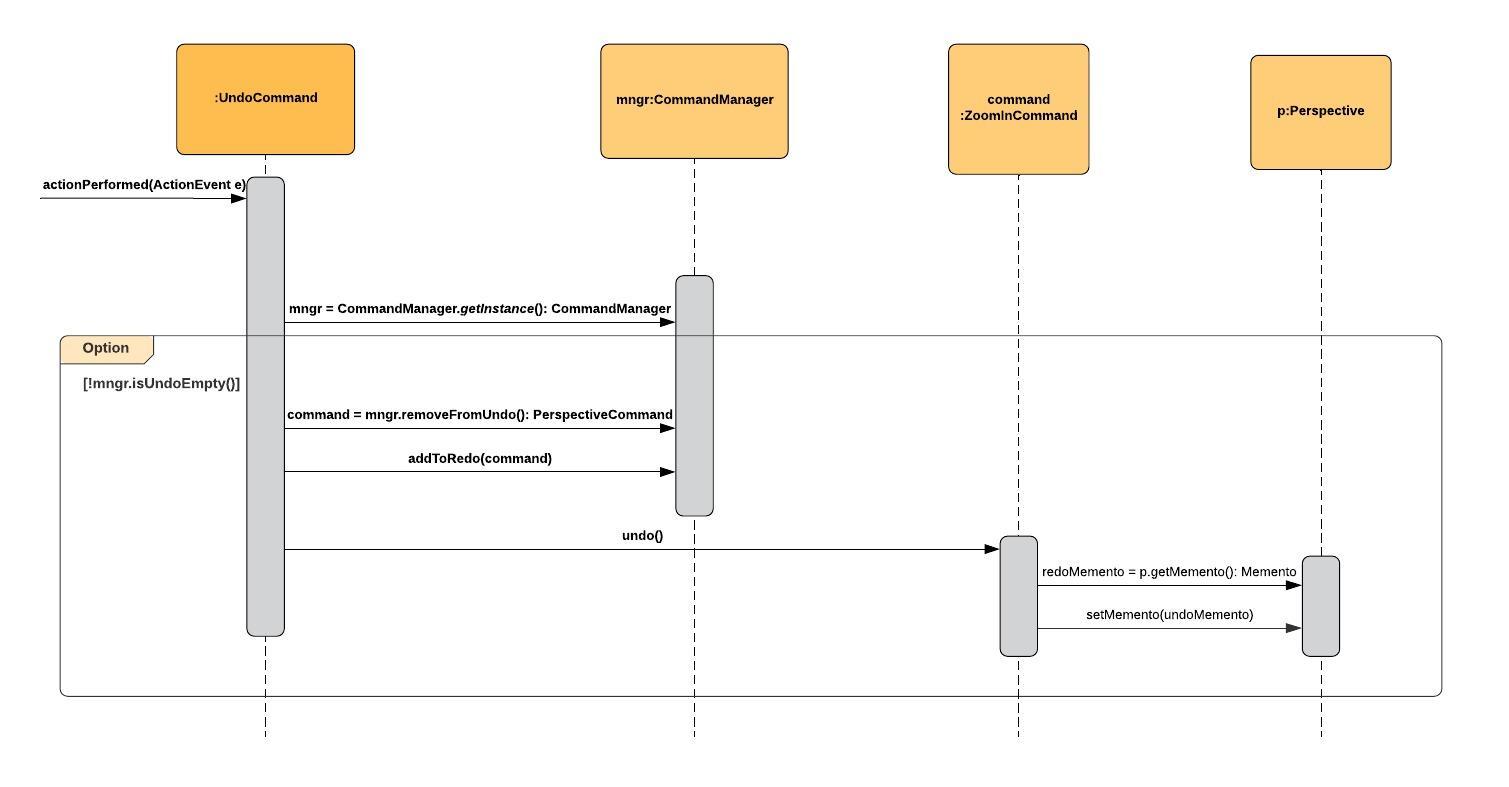
### 2.5.1. Dynamique de l'architecture MVC : effectuer une certaine commande



Détails :

Voici le diagramme de séquence de l’exécution d’une commande, une ZoomInCommand dans cet exemple. D’abord, une action quelconque sur une vue est écoutée par un écouteur qui déclenche l’appel de execute() de ZoomInCommand. Cette commande va récupérer l’instance unique du CommandManager et s’ajoute à sa liste de commandes qu’il est possible de défaire. Ensuite, cette commande récupère l’état actuel de la Perspective et l’enregistre dans son attribut undoMemento, qui servira si l’action est défaite. ZoomInCommand fait ensuite appel à sa méthode executeAction(), qui appelle la méthode zoomIn(int incrementZoom) de la Perspective et modifie son attribut zoom.

### 2.5.2. Dynamique de l'architecture MVC : défaire une certaine commande



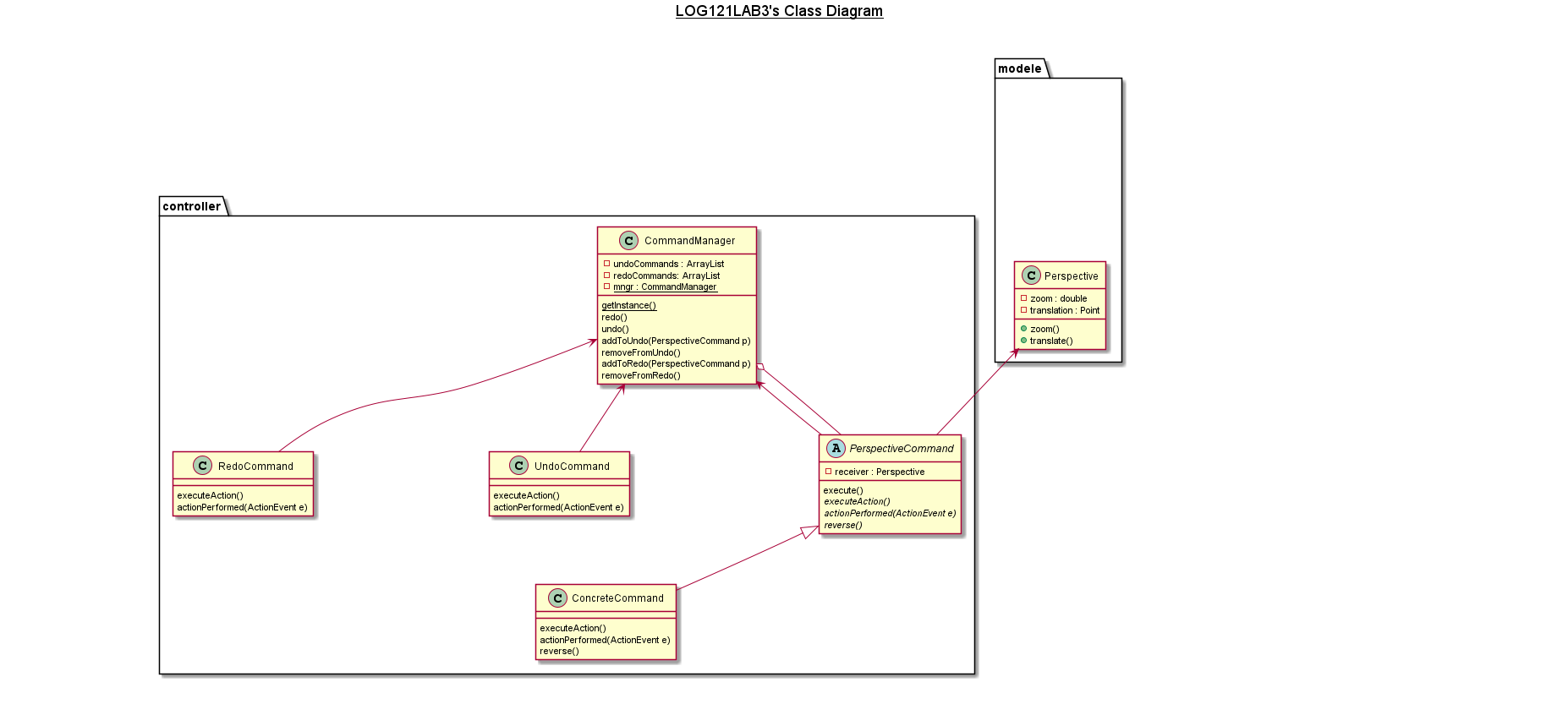
Détails :

Voici l’exemple d’une séquence déclenchée lorsqu’on défait une commande, une ZoomInCommand dans cet exemple. D’abord, UndoCommand écoute un bouton qui est enclenché et appelle sa méthode actionPerformed(). Ensuite, UndoCommand récupère l’instance unique du CommandManager et si sa liste de commandes à défaire n’est pas vide, la commande en tête de liste est retirée et récupérée. Cette commande est alors ajoutée à la liste de commandes qu’on peut refaire du CommandManager. UndoCommand appelle ensuite la méthode undo() de la commande récupérée, qui est dans notre exemple une ZoomInCommand. Finalement, la ZoomInCommand va faire de l’état actuel de la Perspective son redoMemento, qui sera utilisé si la commande est refaite, et change ensuite la Perspective pour l’état enregistré dans son undoMemento.

# 3 Décision de conception / d'implémentation

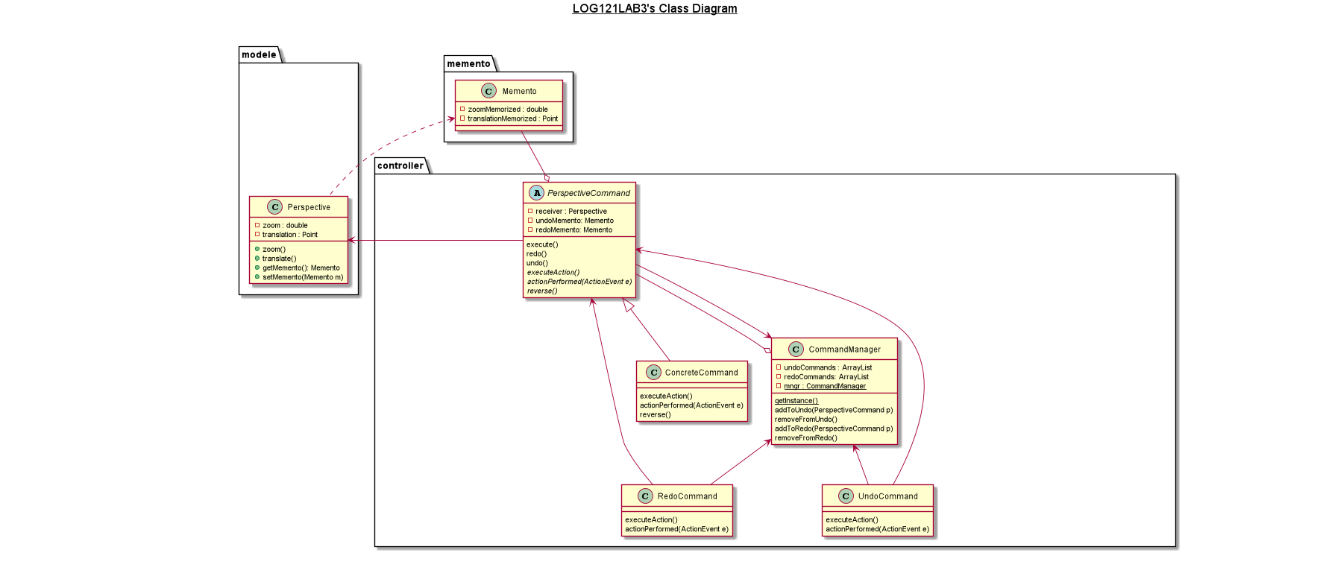
## 3.1 Décision 1 : La Façon dont on défait ou défait une commande

* **Contexte**: La conception de notre application nécessite de défaire des commandes pour revenir en arrière. Plusieurs solutions s’offrent à nous en tant que développeurs pour pouvoir implémenter cette fonctionnalité.
* **Solution 1**: Utiliser des listes de commandes que l’on peut défaire ou refaire et une fonction qui inverse les commandes

****

Explication : Dans cette solution, le CommandManager contient une liste de PerspectiveCommands qu’il peut défaire, une autre pour celles qu’il peut refaire. Chaque PerspectiveCommand contient une fonction reverse(), qui retourne une PerspectiveCommand inverse. Par exemple, si une commande fait un zoom de +5%, sa fonction reverse() retournera une commande qui fait un zoom de -%5. Un appel de UndoCommand va chercher la première commande de la liste undoCommands du CommandManager, crée son inverse, l’exécute et ajoute la commande originale à la liste redoCommands. Un appel à RedoCommand va exécuter la première commande de cette liste. Les avantages de cette conception est que nous conservons un nombre bas de classes, tout en limitant grandement le couplage, comme on peut voir dans le diagramme ci-haut où il y a peu de dépendances. Les désavantages sont la prolifération des objets, car chaque UndoCommand va entraîner la création d’une nouvelle commande. Aussi, il y a un défaut de cohésion, car le CommandManager doit implémenter des commandes undo() et redo(), alors que sa fonction devrait être seulement d’enregistrer les commandes.

* **Solution 2** : Utiliser des listes de commandes que l’on peut défaire ou refaire et le patron Memento

****

* Explication : Dans cette conception, encore une fois, le CommandManager contient une liste de PerspectiveCommands qu’il peut défaire, une autre pour celles qu’il peut refaire. Toutefois, chaque PerspectiveCommand contient son undoMemento et possiblement son redoMemento, qui sont respectivement les états de la Perspective qui sont sauvegardés avant l’exécution d’une commande ou avant de défaire une commande. Donc, si UndoCommand est exécuté, on va récupérer la première commande de la liste undoCommands du CommandManager, et on met la Perspective à l’état du undoMemento de cette commande. L’état initial est sauvegardé dans le redoMemento de cette commande, qui est ajoutée à la liste redoCommands du CommandManager. Cette conception amène plusieurs avantages. Elle permet de diminuer le nombre d’objets, car la même instance de ConcreteCommand se déplace d’une liste à l’autre, au lieu de créer de nouvelles commandes à chaque fois qu’on défait une commande. Aussi, le couplage y est diminué au maximum, et il est très cohésif que la classe PerspectiveCommand implémente undo() et redo(), car ce sont des commandes. Ces deux méthodes permettront à chaque ConcreteCommand, par un appel polymorphique, d’appliquer les changements nécessaires aux listes du CommandManager tout en appliquant leur Memento à la Perspective. Un désavantage de cette conception est que UndoCommand et RedoCommand dépendent maintenant de CommandManager et de PerspectiveCommand.
* **Choix de la solution et justification :**

Nous avons choisi la solution 2, soit l’implantation du patron Memento. Son utilisation très efficace du polymorphisme qui permet aux commandes de modifier les listes du CommandManager, sa cohérence dans la répartition des tâches entre les classes ainsi que la sauvegarde des états qui permet de diminuer grandement le nombre d’objets ConcreteCommand crées ont fait pencher la balance de son côté. Ces avantages éclipsent grandement la dépendance supplémentaire de UndoCommand et RedoCommand.

# 4 Conclusion

Dans ce laboratoire, nous devions utiliser les différentes notions de patrons vus en classe afin de réaliser une application de modification de perspective d’image, permettant l’affichage d’une image sur une interface graphique, l’application de différentes commandes dessus (translation, zoom, défaire et refaire) ainsi que la sauvegarde de ces modifications. L’ensemble de ces objectifs a été atteint : nous avons pu utiliser différents patrons en plus de ceux requis dans la consigne (Command, Singleton, Observer, mais aussi Memento). Nous avons fait en sorte que les commandes soient modulables dans ce projet, afin de rajouter plus facilement de nouvelles commandes si besoin, mais il est possible que nous ayons ajouté trop de classes en voulant rendre ce projet le plus modulable possible, ce qui impacterait sur la lisibilité et la clarté des classes.

Nous avons déjà permis dans l’application l’utilisation des commandes Zoom et Translation à l’aide de la souris directement sur la photo (zoom avec la roulette et translation avec un drag&drop), mais nous pourrions aussi permettre d’utiliser certaines commandes en utilisant une combinaison de touches du clavier (notamment Ctrl-Z pour défaire, Ctrl-O pour ouvrir une image, Ctrl-S pour enregistrer…) afin de permettre une manière maniabilité pour l’utilisateur dans l’application.

Enfin, nous pourrions aussi simplement ajouter de nouvelles fonctionnalités pour cette application (Tourner l’image, la retourner… beaucoup de possibilités s’offrent à nous).

# 5 Références

* Tutorials Point (2020) Java – Serialization. Repéré à <https://www.tutorialspoint.com/java/java_serialization.htm>
* Java2S (2020) Creating a Image Zoomer using Graphics2D : BufferedImage « 2D Graphics « Java Tutorial. Repéré à <http://www.java2s.com/Tutorial/Java/0261__2D-Graphics/CreatingaImageZoomerusingGraphics2D.htm?fbclid=IwAR3H0XkaU_pOumKDNQK8ZnWk0o_uJ4OUuJUfoHVft499EXsNoQh_f14OVcA>
* Java2S (2020) Translating a Drawn Image : Image « 2D Graphics « Java Tutorial. Repéré à <http://www.java2s.com/Tutorial/Java/0261__2D-Graphics/TranslatingaDrawnImage.htm?fbclid=IwAR241Z8s8-yG2od6n7G_T7ukHwBNkl0tpw_I-ZULeOk7k0Zueej5d_9MuHc>