Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

N° de laboratoire	Laboratoire 4
Étudiant(s)	Olivier Nadeau, Mikaël Sauriol, Christelle Sissoko, Louis-Simon Mc Nicoll
Code(s) permanent(s)	NADO26058604,SAUM31059607, SISC24579504, MCNL14129509
Cours	LOG121
Session	Hiver 2017
Groupe	02
Professeur	Francis Cardinal
Chargés de laboratoire	Antoine Grenier, Mathieu Ouellet
Date de remise	13 avril 2017

1 INTRODUCTION

Dans ce dernier laboratoire de LOG121, nous avions comme tâche à réaliser d'implémenter une application contenant plusieurs perspectives où l'utilisateur va pouvoir « zoomer » et effectuer des translations sur l'image. De plus, l'utilisateur aura aussi l'option de sauvegarder et de charger d'un menu fichier l'état d'une perspective modifiée. L'architecture MVC devient alors très efficace dans un laboratoire comme celui-ci puisque seul un contrôleur va pouvoir gérer les différentes commandes sur chacune des vues. Le fait d'utiliser cette méthode de conception peut permettre, dans le futur, d'ajouter des commandes ainsi que des vues sans avoir à modifier tout le code.

Les objectifs de ce laboratoire sont d'implémenter une application selon l'architecture Modèle/Vue/Contrôleur (MVC) afin de pouvoir gérer les différentes vues, d'utiliser le patron Command afin de défaire les opérations (undo) et d'appliquer le patron singleton afin d'avoir une instance unique en ce qui concerne la mémorisation des différentes commandes.

En lisant attentivement ce rapport, vous verrez que les objectifs de ce laboratoire ont été atteints et bien compris par notre équipe. Premièrement, vous trouverez dans la section Conception les choix de conception ainsi que des représentations graphiques tels que des diagrammes de classes et de séquences qui vous aideront à mieux comprendre le fonctionnement de notre application tout en mentionnant les faiblesses de celle-ci. Par la suite, les décisions d'implémentation seront abordées ou on y présentera nos solutions finales et pourquoi nous avons décidé d'adopter une solution plutôt qu'une autre. Enfin, vous arriverez à la conclusion où nous avons fait une synthèse et une recherche sur de nouvelles informations. Pour consulter nos sources, allez à la section références.

2 CONCEPTION

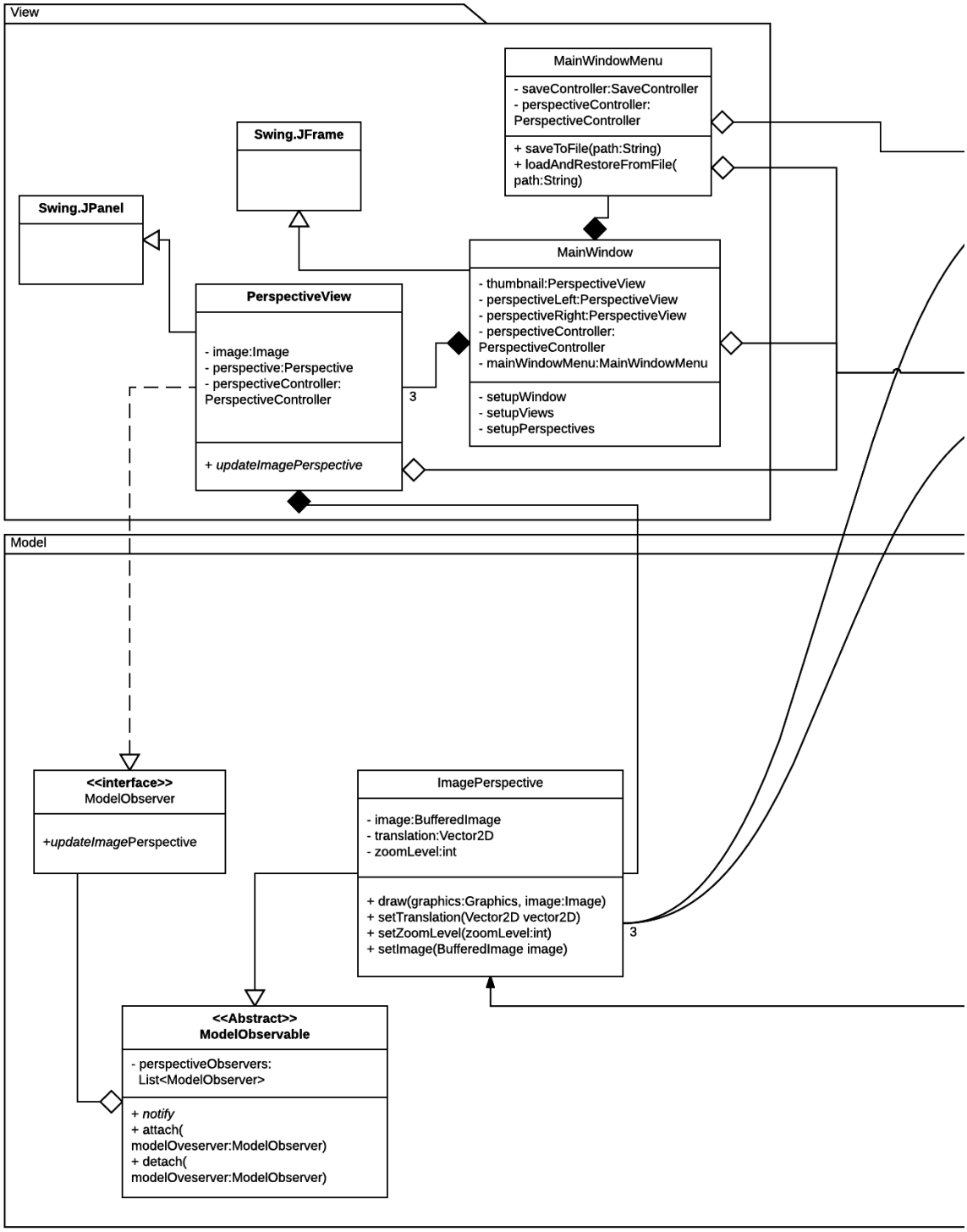
2.1 CHOIX ET RESPONSABILITÉS DES CLASSES

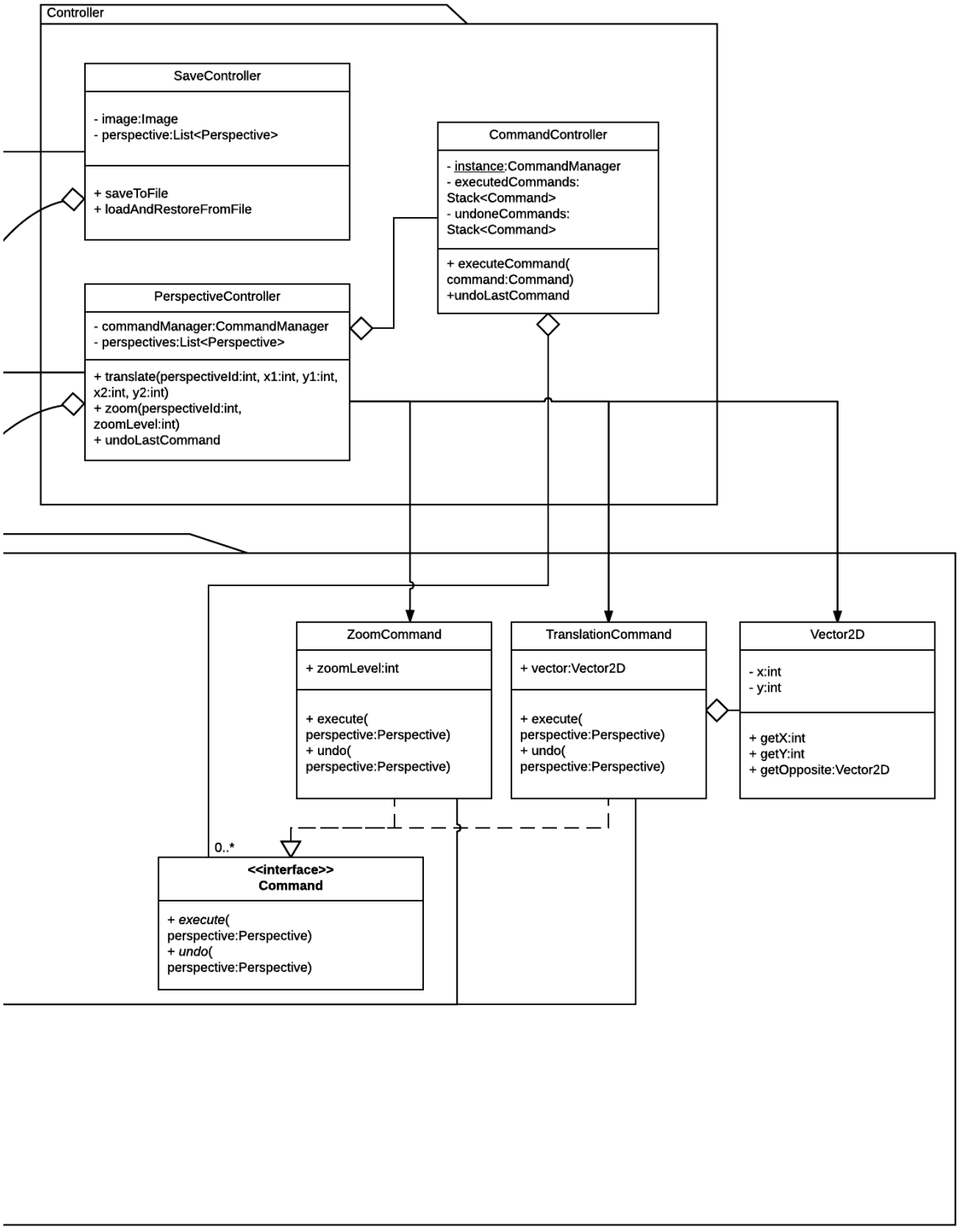
Classe	*Responsabilités*	*Dépendances*
PerspectiveView	Cette classe affiche les perspectives de l'application (image)	– Hérite de JPanel – Implémente ModelObserver
MainWindow	Cette classe affiche l'interface principale de l'application qui est découpée en 3 vues	– PerspectiveView – MainWindowMenu
MainWindowMenu	Cette classe crée le menu de l'application	–
ModelObserver	Cette interface fournie une méthode updateImagePerspective	– ModelObservable
Image	Cette classe permet de définir toutes les caractéristiques d'une image	– Hérite de ModelObservable – PerspectiveView – Perspective
Perspective	Cette classe permet de définir toutes les caractéristiques liées à la perspective	– Hérite de ModelObservable – Image – PerspectiveView

		– SaveController – PerspectiveController
ModelObservable	Cette classe abstraite contient les méthodes liées aux observateurs	–
SaveController	Cette classe sert à enregistrer une image et une perspective	– MainWindowMenu
PerspectiveController	Cette classe sert à appliquer la translation et le zoom faite sur la perspective	– Vector2D – TranslationCommand – ZoomCommand – PerspectiveView – MainWindow – MainWindowMenu
CommandController	Cette Classe sert à faire la gestion des commandes de l'application	– PerspectiveController
ZoomCommand	Cette classe permet de définir tout ce qui a rapport au zoom	– Implémente Command – Perspective
TranslationCommand	Cette classe permet de définir tout ce qui a rapport à la translation	– Implémente Command – Perspective
Vector2D	Cette classe permet de définir la	– TranslationComm

	pasition en x et y et fourni son opposé	and
Command	Cette interface fournie les méthodes pour exécuter une commande et pour l'annuler	– CommandControll er

2.2 DIAGRAMME DES CLASSES




Controller
SaveController
- image:Image
- perspective:List<Perspective>
+ saveToFile
+ loadAndRestoreFromFile
CommandController
- instance:CommandManager
- executedCommands:
Stack<Command>
- undoneCommands:
Stack<Command>
+ executeCommand(
command:Command)
+undoLastCommand
PerspectiveController
- commandManager:CommandManager
- perspectives:List<Perspective>
+ translate(perspectiveId:int, x1:int, y1:int,
x2:int, y2:int)
+ zoom(perspectiveId:int,
zoomLevel:int)
+ undoLastCommand
ZoomCommand
+ zoomLevel:int
+ execute(
perspective:Perspective)
+ undo(
perspective:Perspective)
TranslationCommand
+ vector:Vector2D
+ execute(
perspective:Perspective)
+ undo(
perspective:Perspective)
Vector2D
- x:int
- y:int
+ getX:int
+ getY:int
+ getOpposite:Vector2D
0..*
<<interface>>
Command
+ execute(
perspective:Perspective)
+ undo(
perspective:Perspective)

2.3. Utilisation des patrons de conception

2.3.1 Patron « Observer »

Patron Observer	**Nom dans le laboratoire**
Observer	ModelObserver
update()	updateImagePerspective()
Observable	ModelObservable
notify()	notify()
attach(Observer)	attach(ModelObserver)
detach(Observer)	detach(ModelObserver)
ConcreteSubjects	Image et Perspective

Nous avons décidé d'utiliser le patron « Observer » puisque dans ce contexte, il y avait plusieurs vues observant l'état de plusieurs modèles. En effet, cette solution permet de mettre à jour les vues dès le changement de perspective en capturant les événements de la souris. Grace à ce patron tout changement est notifié et les perspectives sont mises à jour.

2.3.2 Patron « Singleton »

Patron Singleton	**Nom dans le laboratoire**
Singleton	CommandManager
getInstance : Singleton	getInstance : CommandManager

Dans ce cas, nous voulions qu'il existe qu'une seule instance de l'objet « CommandManager » afin de garantir qu'il n'y ait qu'une seule liste de commandes à annuler.

2.3.3 Patron « Command »

Patron Command	**Nom dans le laboratoire**
Command	Command
execute()	execute(), undo()

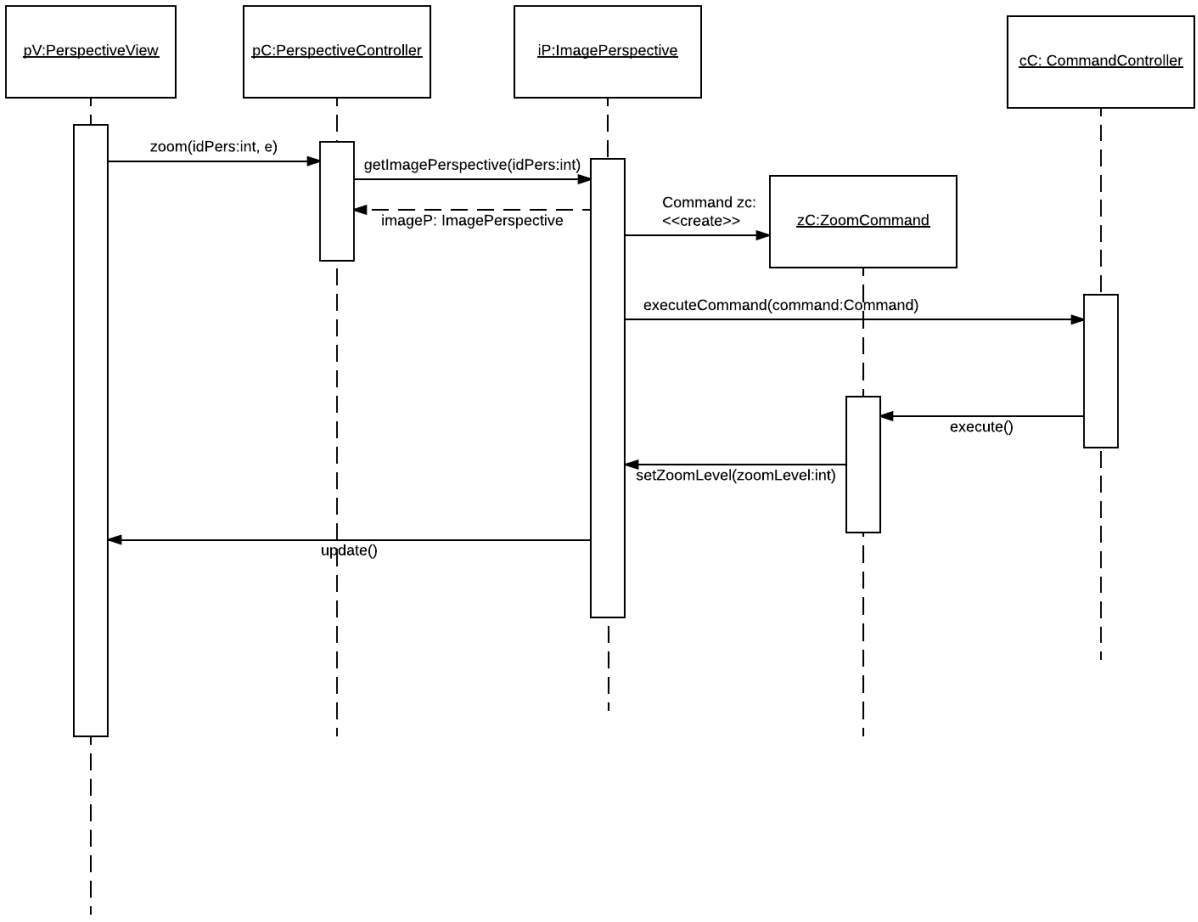
Nous avons opté pour le patron de conception « Command » parce qu'il permet de séparer le code qui initie l'action de l'action elle-même. Par exemple, dans notre code l'action « undo » ou « redo » est connecté à une commande sans savoir ce que fait cette commande. Ce patron permet d'intégrer plusieurs commandes et évite la prolifération de méthode. Ce qui rend le code plus facile à maintenir.

2.4 Faiblesses de la conception

La première faiblesse est l'annulation des commandes de zoom. Celles-ci sont en effet sauvegardées en tant que petits incréments. Ainsi lorsque l'on annule un gros zoom on n'annule en fait qu'une partie de celui-ci. Il aurait fallu implémenter une façon de regrouper plusieurs petits zooms en une seule commande zoom plus grande. Une autre faiblesse serait que la vue servant de référence et qui ne peut être modifiée, nommée le « thumbnail », est en fait la même classe que celles pouvant être modifiées. La seule chose empêchant cette vue d'être utilisable est que ses évènements sont désactivés. La « thumbnail » devrait être plus séparé thématiquement des perspectives. Si son comportement venait à changer davantage il faudrait alors l'implémenter dans une autre classe. L'image et la perspective ont été fusionnées. Bien que cette décision présente des avantages, si l'on avait à implémenter d'autres fonctionnalités liées à l'image alors il faudrait ramener l'implémentation de l'image séparée et on aurait perdu notre temps. Ainsi garder les classes séparées les rend plus flexibles tandis que les regrouper les rend plus aisées à manier.

2.5. Diagramme de séquence (UML)

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

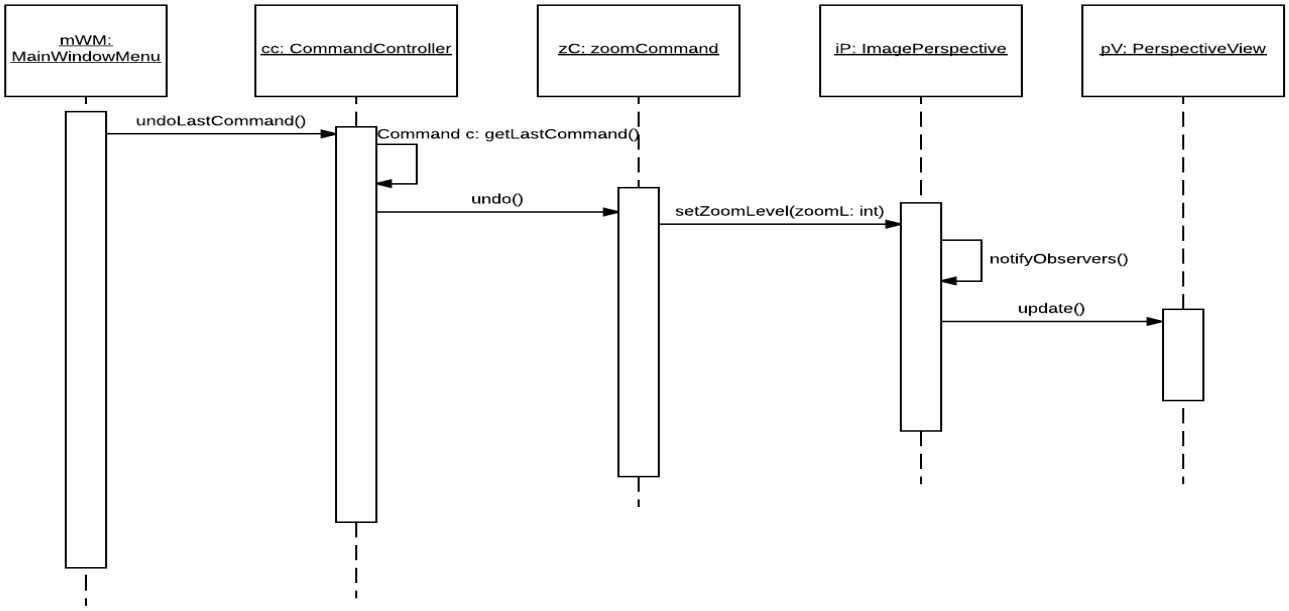


Exemple de la commande « zoom »

Un évènement est déclenché à partir de la perspective. Lorsque l'évènement est déclenché, la commande « zoom » du contrôleur des perspectives est appelé en lui passant en paramètres le numéro d'identification de la perspective afin de savoir sur laquelle l'évènement devait être appliqué. Une fois la perspective récupérée grâce à son numéro d'identification, la commande zoom est exécutée dans « CommandController ».

Une fois exécutée, le niveau de zoom de la perspective est mis à jour et par la suite la vue est « updatée »

2.5.2. DYNAMIQUE DE L'ARCHITECTURE MVC : DÉFAIRE UNE CERTAINE COMMANDE



Lorsque l'utilisateur clique sur le menu « Undo » de la barre de tâche, l'évènement déclenché appelle la commande « undoLastCommand » du contrôleur « CommandController ». Le contrôleur récupère la dernière commande emmagasinée dans sa banque de commandes. Une fois la dernière commande récupérée, la méthode « undo » est appelée. Ensuite, le niveau de zoom est mis à jour dans la perspective correspondant au dernier évènement. Tous les observateurs sont ainsi mis à jour et la vue est « updatée » en conséquence.

3 DÉCISION D'IMPLÉMENTATION

3.1 DÉCISION 1 : IDENTIFIER LES PERSPECTIVES À L'AIDE D'UN INDENTIFIANT

- **Contexte**: Nous devions trouver une solution afin de différencier les trois perspectives.
- **Solution 1**: Ajouter comme paramètre un Int qui les différencierait dans la méthode setupViews ().

```
private void setupViews() {
    thumbnail = new PerspectiveView( idPerspective: 0,  occupiedWidthRation: 0.5, perspectiveController);
    perspectiveViewLeft = new PerspectiveView( idPerspective: 1,  occupiedWidthRation: 1, perspectiveController);
    perspectiveViewRight = new PerspectiveView( idPerspective: 2,  occupiedWidthRation: 1, perspectiveController);
```

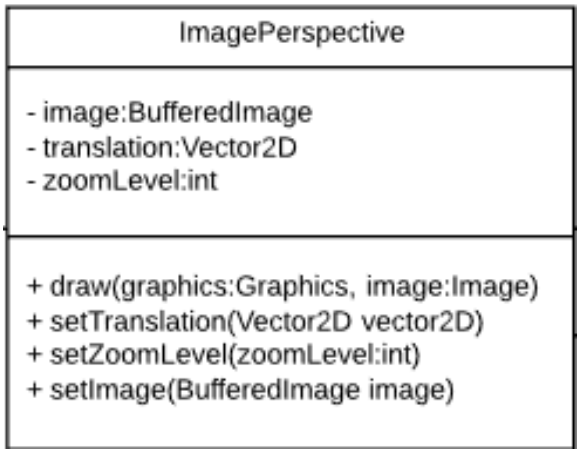
- **Solution 2** : Envoyer la perspective elle-même aux contrôleurs dans la méthode setupViews().

```
public PerspectiveController(ImagePerspective perspective1, ImagePerspective perspective2,
                             ImagePerspective perspective3) {
    this.perspective1 = perspective1;
    this.perspective2 = perspective2;
    this.perspective3 = perspective3;
}
```

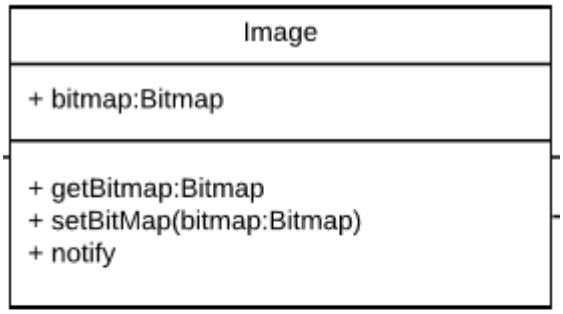
- **Choix de la solution et justification :** Nous avons opté pour la solution 1 puisqu'il était facile de différencier les perspectives par rapport à leur numéro assigne, donc dans l'initialisation de chaque vue, nous avons ajoute un paramètre. Le seul défaut que cela apporte est lorsqu'on appelle une méthode, le code doit toujours passer par un if, qui vérifie si la perspective est l'une des deux qui est modifiable. Dans la solution 2, le fait de passer chaque vue au contrôleur créer beaucoup de couplage entre les classes, donc si l'on voudrait améliorer le programme dans le futur ainsi que d'ajouter des fonctionnalités ou bien des vues, il y aurait beaucoup plus de modification de code à faire dans la solution 2 que dans la solution 1.

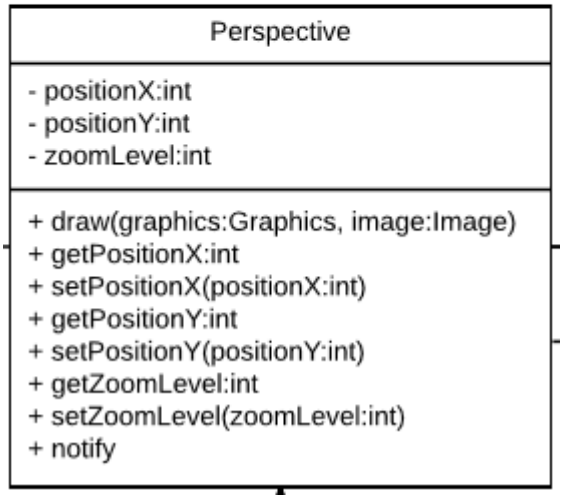
3.1 DÉCISION 2 : FUSIONNER LES CLASSES IMAGES ET PERSPECTIVES

- **Contexte** : La gestion des perspectives avec deux classes devenait difficile à maintenir.
- **Solution 1**: Fusionner les perspectives et l'image



- **Solution 2** : Garder les deux classes séparer





- **Choix de la solution et justification :** Nous avons opté pour la solution 1 puisque fusionner les deux classes ensemble rendrait l'implémentation plus facile et la maintenance de la classe se ferait plus rapidement puisque l'on n'a pas besoin de modifier deux classes. Par contre, la solution 2 présente plus d'avantages à long terme puisque la flexibilité des choix est plus disponible que la solution 1 et que l'implémentation est plus efficace puisque les deux classes sont propres a elles-mêmes. Bref, la solution 1 est plus efficace dans notre cas puisque dans un court délai, nous devions remettre le laboratoire.

4 CONCLUSION

En conclusion, nous pensons que l'application répond bien aux besoins du client. Le projet nous a permis de nous familiariser avec l'utilisation des patrons de conception tel que « Observer », « Singleton » et « Command ». De plus, nous avons pu nous servir de l'architecture MVC pour bien mettre à terme notre application. Dans la décision d'implementation1 qui consistait à trouver une façon de différencier les perspectives, nous avons opté pour la solution 1, qui était de passer un paramètre Int à chaque perspective. Pour la décision d'implémentation 2, la gestion des perspectives avec deux classes devenait difficile à maintenir, donc la solution finale était de fusionner les classes images et perspective afin de créer Image Perspective, qui rendrait l'implémentation à court terme plus facile. Pour ce qui est des faiblesses de conception, la faiblesse la plus flagrante sont les zooms, qui se font un à la fois, donc lors du undo, chaque petit zoom devient un undo. Bref, nous pensons avoir bien répondu à l'énoncé du laboratoire et pensons que ce petit laboratoire pourrait être utile dans un futur ou l'on voudrait avoir notre propre zoom sur n'importe quelle plateforme de programmation, alors rendre le code universel.

5 RÉFÉRENCES

Ouvrage :

- Horstmann, C., Object-Oriented Design and Patterns. Second Edition, Wiley, 2006