Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

N° de laboratoire	Laboratoire 3
Étudiant(s)	Bakaleinik, Yulia Fily, Donatien Palashev, Valentin Quezada, Jonnie klein Racanelli, Anthony
Code(s) permanent(s)	BAKY30539705, FILD92110006, PALV04049901, QUEJ19129809, RACA28079504
Cours	LOG121
Session	Automne 2021
Groupe	04
Professeur	**Benoit Galarneau**
Chargés de laboratoire	**Bianca Popa**
Date de remise	Le 30 novembre 2021

1 INTRODUCTION

L'utilisation de patrons de conceptions dans l'élaboration de n'importe quel logiciel est primordiale. Cependant, certains de ces patrons ne sont pas facilement compréhensibles et nécessitent beaucoup de travail afin de les maitriser. Ce laboratoire est une occasion parfaite pour mieux comprendre certains de ces patrons, et par la même occasion, développer une application fonctionnelle, utile et évolutive.

L'objectif de ce laboratoire est de concevoir et d'implémenter un logiciel permettant la manipulation d'une image. Nous devons donc être en mesure d'effectuer un changement d'échelle et de positionnement et de pouvoir annuler un ou plusieurs changements. L'utilisateur du logiciel devra aussi avoir la possibilité de sauvegarder ses changements. La sauvegarde pourra par la suite être utilisée afin de continuer la modification du fichier. Notre mandat est de respecter ces objectifs tout en utilisant les opérations effectuées avec la souris.

Dans ce rapport, nous débuterons avec la présentation des choix et responsabilités de nos classes nécessaires au fonctionnement de notre logiciel. Notre conception sera plus amplement expliquée à l'aide d'un diagramme de classe UML. Ensuite, une analyse des faiblesses de conception sera présentée avec de potentielles solutions. Cette présentation sera suivie par l'affichage de diagrammes de séquence illustrant et expliquant la dynamique de l'architecture MVC lorsqu'une commande est effectuée et lorsqu'une commande est défaite. Par la suite, nous poursuivrons avec l'explication de notre décision concernant la gestion des commandes effectuées par l'utilisateur. Nous conclurons avec un résumé des points forts et des points faibles de notre conception et des possibilités de développements futurs.

2 CONCEPTION

2.1 CHOIX ET RESPONSABILITÉS DES CLASSES

Classe	*Responsabilités*	*Dépendances*
CommandCenter	Facilite l'utilisation du patron Command en centralisant toutes ses utilisations.	ICommand CommandType
Redimensionner	Permets de redimensionner la taille de l'image	ICommand
Annuler (undo)	Permets de faire revenir l'image à un état précédent.	ICommand
Enregistrer	Permets d'enregistrer une image et son état.	ICommand
Translation	Permets de modifier les coordonnées d'une image.	ICommand
Perspective	Contiens une *Image* et permets de modifier sa position.	ISaveElement IObserver Image
Image	Contiens une *BufferedImage* et permets de modifier ses dimensions.	IOriginator ISaveElement
ImageSnapshot	Permets de créer une sauvegarde de l'état de l'image.	IMemento
PerspectiveSnapshot	Permets de créer une sauvegarde de l'état de la perspective.	IMemento
History	Permets de sauvegarder les différents états d'une image.	IMemento
ImageUtils	Permets d'ajouter une image dans l'application	ISubject

2.2 Diagramme des classes

Voir class-diagram.png

2.3. Utilisation des patrons de conception

2.3.1 Patron Observateur

Ce patron est beaucoup utilisé au sein de l'application. Les deux classes principales l'utilisant sont *Image* et *Perspective*. Les deux l'utilisent de façon similaire. Perspective lorsque les coordonnées sont modifiées et Image lorsque sa taille est modifiée. Cela permet d'établir une connexion entre plusieurs parties du code facilement sans surcharger et sans avoir de problèmes liés au couplage.

Classe par défaut	**Classe dans le laboratoire**
ConcreteSubject	Image Perspective
Méthodes par défaut	**Méthodes dans le laboratoire**
addObserver(Observer) notifyObservers() removeObserver(Observer)	attach(Observer) notifyObservers(EventType, Object) detach(Observer)
Attribut par défaut	**Attribut dans le laboratoire**

2.3.2 Patron Commande

Le patron commande est utilisé afin de gérer efficacement les différentes actions réalisables sur une Image. Cela facilite donc l'utilisation de *Redimensionner, Translation, Undo, Redo* et *Enregistrement.* La classe la plus utilisée dans ce laboratoire est CommandCenter. Cette classe centralise l'ensemble des commandes possibles au sein de l'application.

Classe par défaut	**Classe dans le laboratoire**
Invoker	CommandCenter
Méthodes par défaut	**Méthodes dans le laboratoire**
executeCommand()	executeCommand()
Attribut par défaut	**Attribut dans le laboratoire**
	BiMap<CommandType, ICommand> commands

2.3.3 Patron Mémento

Plusieurs classes définissent le patron de conception Memento. La principale étant History.
Cette classe est utilisé afin de stocker de snapshots, soit d'image, soit de perspectives.
Ces snapshots sont créés grâce à *ImageSnapshot* ou *PerspectiveSnapshot*.
Cet historique est ensuite utilisé afin de pouvoir annuler des changements effectués (undo).

Classe par défaut	**Classe dans le laboratoire**
CareTaker	History
Méthodes par défaut	**Méthodes dans le laboratoire**
add(Memento) get(Memento) ----	pushSnapshot(Memento) ---- undo()
Attribut par défaut	**Attribut dans le laboratoire**
List<Memento> mementoList	Stack<Memento> history

2.3.4 PATRON SINGLETON

Le patron singleton permet de travailler sur une instance unique afin d'éviter que plusieurs objets soient créés. CommandCenter utilise ce patron, car il est impératif pour cette classe d'être unique. Cela facilite également l'accès à cette instance.

Classe	**Nom dans le laboratoire**
Singleton	CommandCenter
Méthode	**Nom dans le laboratoire**
getInstance()	getInstance()
Attribut	**Nom dans le laboratoire**
Singleton singletonInstance	CommandCenter INSTANCE

2.3.5 PATRON VISITEUR

Le patron visiteur a été utilisé afin de récupérer et modifier les informations relatives aux images des deux panneaux principaux. Il est majoritairement utilisé lorsqu'un changement est effectué.

Classe	**Nom dans le laboratoire**
ConcreteVisitor	UndoVisitor
Méthode	**Nom dans le laboratoire**
visit(ConcreteElement)	visit(Perspective)
Attribut	**Nom dans le laboratoire**

2.3.6 PATRON MÉDIATEUR

Le patron médiateur permet d'encapsuler plusieurs objets afin d'avoir des interactions plus simples entre différentes parties du code et de diminuer le couplage. Au sein du logiciel, plusieurs médiateurs sont utilisés pour les différentes commandes afin de faciliter leurs utilisations.

Classe	**Nom dans le laboratoire**
ConcreteMediator	TranslationMediator
Méthode	**Nom dans le laboratoire**
notify(sender)	notify(sender, Objet payload)
Attribut	**Nom dans le laboratoire**

2.4 Faiblesses de la conception

2.4.1 PanneauImageBas & PanneauImageTop

La conception de deux classes, <PanneauImageBas> et <PanneauImageTop> utilisent presque intégralement les mêmes méthodes et attributs, ce qui rend le code répétitif. Le fait de séparer ces deux classes nous permet de manipuler les images se trouvant sur les deux panneaux principaux plus aisément. Une possible solution serait de fusionner ces classes afin qu'il n'y en ait plus qu'une. Grâce à cette fusion, nous pourrions utiliser deux instances. Chaque instance serait alors attribuée à un panneau différent afin de ne pas surcharger notre application. De ce fait, les manipulations peuvent être effectuées dans chaque panneau. De plus, cela permettrait d'ajouter facilement d'autres panneaux ayant la même fonction au sein de l'application.

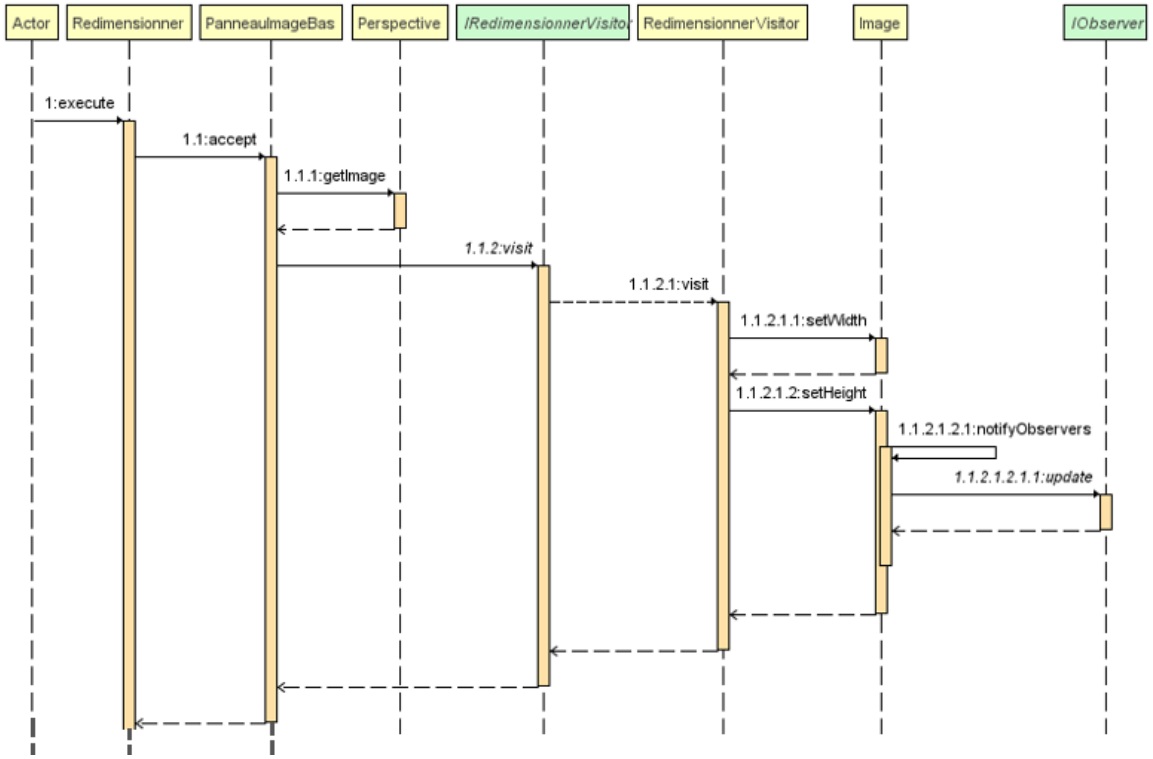
2.4.2 Les variables ont parfois besoin de traverser trop de couches de panneaux Java Swing (JPanel)

L'application comporte également un problème lié à la conception des différents panneaux, ce qui provoque des complications lors de l'écriture du code. Par exemple, lors d'une action refaire (redo), la commande doit passer par le menu déroulant, puis par le panneau. Cela crée une complexité de conception car les variables ont besoin de traverser trop de couches de JPannel afin d'être exécutées avec succès. Des commandes peuvent parfois nécessiter beaucoup de ressources. Cette faiblesse n'étant pas une bonne pratique de conception, il faudrait alors y remédier. Il serait alors plus pratique de centraliser la création du cadre JPannel. Cela pourrait être réalisable en faisant en sorte que le cadre crée ses propres panneaux et éléments afin que les références soient plus faciles d'accès.

2.5.Diagramme de séquence (uml)

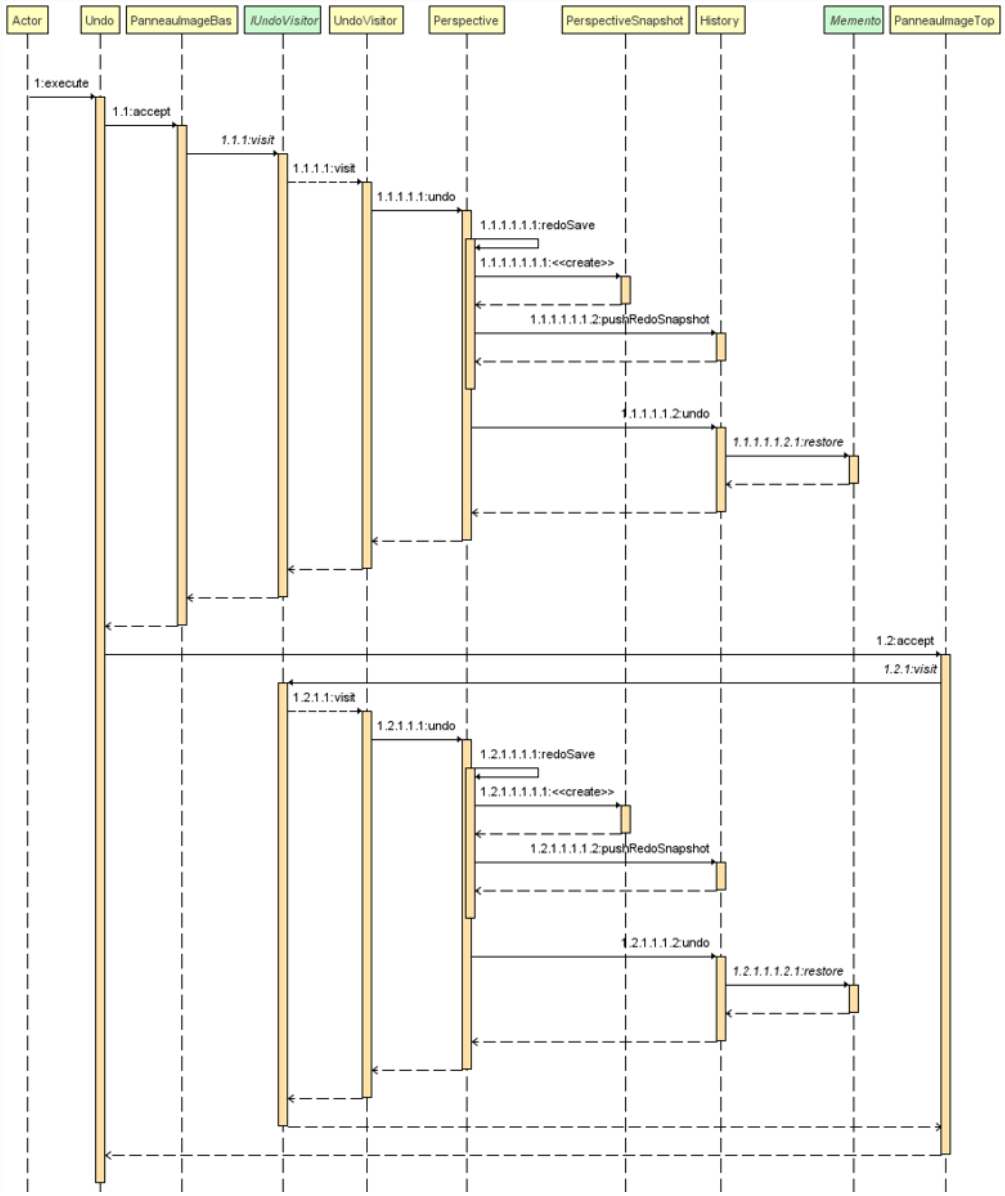
2.5.1. DYNAMIQUE DE L'ARCHITECTURE MVC : REDIMENSIONNER UNE IMAGE

Lorsque l'utilisateur désire redimensionner une image qui se trouve dans le panneau inférieur ou supérieur, il sera redirigé vers le patron visiteur qui prendra en compte des dimensions de l'image initiale afin d'augmenter ou de diminuer sa taille. Finalement tout changement sera notifié et actualisé grâce au patron observateur.



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

Lorsque la commande undo est appelée par le centre de commande pour défaire la commande précédente, dans le panneau d'image inférieur ou supérieur ; celle-ci est directement redirigée vers le patron visiteur. De ce point, grâce à la perspective de l'image et l'historique de celle-ci, nous pouvons utiliser la commande undo qui retournera l'image précédente grâce au patron memento.



3 Décisions de conception / d'implémentation

3.1 Décision 1 : Gestion des commandes effectuées par l'utilisateur

- **Contexte**:

  Dans cette application, il est nécessaire d'y avoir un flux d'exécution structuré pour l'exécution des ICommands. Le côté client doit être capable d'exécuter des commandes indépendamment de son implémentation. En d'autres termes, l'implémentation d'une action doit être encapsulée ou abstraite pour simplifier l'utilisation au niveau de l'utilisateur.

- **Solution 1**: Il est possible de créer une classe centrale appelée <CommandCenter>. Cette classe a comme objectif d'agir comme une télécommande pour l'ensemble des opérations qu'un usager peut effectuer. <CommandCenter> possède une liste de commande et utilise le patron de conception Singleton, ce qui rend l'exécution des commandes beaucoup plus simple. De plus, cela permet d'accéder à l'instance de cette classe partout dans le code.

- **Solution 2** : La deuxième solution serait de décentraliser chaque instance d'une classe implémentant ICommand. Cela signifierait que les classes auraient accès à l'ICommand correspondant à leurs responsabilités au lieu d'avoir accès à tous les ICommands. Cela se ferait par injection de dépendances, ce qui impliquerait de structurer le code de manière que les constructeurs permettent l'acceptation d'une ICommand.

- **Choix de la solution et justification :** La première solution est celle choisie, car l'exécution du logiciel est bien plus simple que la deuxième. En effet, l'implémentation de la classe <CommandCenter> permet à notre application d'utiliser une classe unique qui rassemble toutes les commandes. Avec la solution 2, pour effectuer une sauvegarde d'une perspective, nous aurions besoin d'une instanciation dans la classe <Perspective> ainsi que dans la classe <Image>. Cependant, grâce à notre choix, ces classes auront déjà accès aux instances nécessaires. De ce fait, la sauvegarde de l'image s'effectuera plus efficacement.

4 CONCLUSION

Pour conclure, le but de ce laboratoire était de concevoir et d'implémenter un logiciel permettant la manipulation d'une image sur sa perspective afin d'établir des changements d'échelle et de positionnement grâce à la souris. Plusieurs opérations devaient être applicables sur l'image, comme redimensionnées ou déplacées. Il était aussi requis d'inclure la possibilité de défaire et refaire certaines actions effectuées sur une image. Nous avons atteint ces objectifs et la solution fournie permet de répondre pleinement aux critères demandés.

L'application créée possède plusieurs points forts. Un point notable serait celui du couplage. L'utilisation de plusieurs interfaces diminue grandement le couplage entre les classes. De plus, beaucoup d'aspects du programme permettent aux utilisateurs d'utiliser l'application facilement, sans se soucier de ce qui se passe en arrière-plan. Ce qui garantit une expérience utilisateur optimale.

Les plus gros points faibles de l'application sont ceux mentionnés dans les faiblesses de conceptions. Les panneaux ne permettent pas une évolutivité importante. L'aspect java swing quant à lui risque de compliquer la tâche au(x) programmeur(s) souhaitant apporter des modifications. Cependant, ces problèmes sont uniquement situés au niveau du *back-end*. Pour l'utilisateur, aucun de ces problèmes ne sera visible.

Une évolution possible à cette application serait d'inclure la possibilité d'intégrer une caméra sur laquelle il serait possible d'effectuer les mêmes commandes que pour les images. De plus, des commandes additionnelles pourraient améliorer l'expérience utilisateur. Il pourrait être intéressant de rentre possible l'enregistrer en format vidéo du déroulement des changements effectués dans l'application en temps réel.