

|  |
| --- |
| **Département du génie logiciel et des technologies de I’information** |

**Rapport de laboratoire**

|  |  |
| --- | --- |
| **No de laboratoire** | Laboratoire 4 |
| **Étudiant(s)** | Aissou Idriss, Murat David, Vuillaume Alexis |
| **Code(s) permanent(s)** | AISI01088901, MURD12099201, VUIA12119301 |
| **Cours** | LOG121 |
| **Session** | Hiver 2014 |
| **Groupe** | 01 |
| **Professeur** | **Francis Cardinal** |
| **Chargés de laboratoire** | **Patrice Boucher** |
| **Date de remise** | Jeudi 3 Avril 2014 |

**1. INTRODUCTION**

Pour ce laboratoire, nous avons pour objectif de concevoir une application permettant d’afficher une image et offrir la possibilité à l’utilisateur de la modifier, soit en changeant son échelle ou bien de la déplacer à l’écran à l’aide de la souris. De plus, l'utilisateur aura toujours la possibilité de défaire ou de ré-appliquer ses modifications grâce aux commandes « CTRL-Z » (undo), « CTRL-Y » (redo). L'utilisateur disposera d’un menu "Fichier" > "Sauvegarder" qui permettra d’enregistrer l'état de son image dans un fichier sur le disque dur.

Dans ce laboratoire, différentes notions seront exploitées, telles que :

* Les diagrammes de classe UML ainsi que les diagrammes de séquences
* L’utilisation des patrons de conceptions : Modèle vue contrôleur, Fabrique, Mémento, Observer, Commande, Singleton.
* L’utilisation de la sérialisation d'objet en Java

Après analyse, conception et réalisation de la solution, celle-ci permet de répondre à l’ensemble des besoins de l’application. Dès lors, notre solution permet à l’utilisateur de pouvoir afficher l’image de son choix dans l’application afin de la modifier. L’utilisateur peut alors changer l’échelle de son image mais aussi de la translater dans la région d’affichage de l’application. De plus, notre solution permet d’offrir à l’usager la possibilité de sauvegarder son image dans un fichier à l’aide d’un menu, de copier une perspective et de la copier sur une autre image et enfin de pouvoir défaire ou de réappliquer une perspective sur une image.

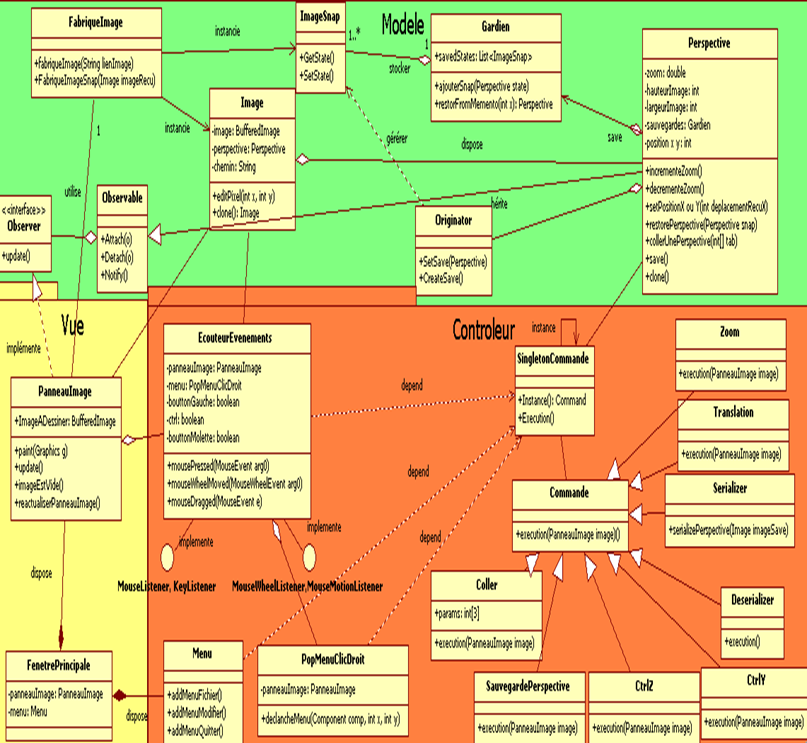
Au cours de ce rapport, nous verrons tout d’abord la description de l’architecture logicielle. Notamment, le choix des responsabilités des classes, le choix des patrons de conceptions ainsi que la faiblesse de l’application. Nous appuierons notre développement à l’aide d’un diagramme UML, et un diagramme de séquence pour comprendre le déroulement de notre application. Enfin, nous évoquerons la décision de l’implémentation.

**2. Description de l'architecture logicielle**

## 2.1. Choix et responsabilités des classes

|  |  |  |
| --- | --- | --- |
| **Classe** | **Responsabilités** | **Dépendances** |
| **Modèle** |  |  |
| **FabriqueImage** | **Application du patron Fabrique** :  **fabriqueImage (String lien) :** méthode qui permet d’instancier une image par utilisation de la classe Image en recevant un paramètre de type chaine de caractère indiquant le lien de l’image.  **fabriqueImageSnap (Image imageRecu)** : méthode qui permet d’instancier une imageSnap (sauvegarde de l’image selon ses transformations antérieures) par l’utilisation de la classe ImageSnap. Celle-ci reçoit en paramètre un type Image. | **PanneauImage** : C’est le panneau d’image situé dans la vue qui utilise la fabrique pour obtenir l’image à afficher.  **Originator :** C’est aussi Originator qui utilise la fabriqueImage afin de retourner une instance d’ImageSnap. |
| **Image** | **Image (String lienImage)** : Constructeur de la classe Image qui va permettre d’instancier une image en recevant en paramètre le chemin de l’image. De plus, c’est ici que se créer une instance de la perspective de l’image (hauteur, largeur.  **Image (Image img)** : Constructeur de la classe Image qui va permettre d’instancier une Image en recevant en paramètre soit une Image ou une ImageSnap. Ainsi, elle va récupérer les données de celle-ci afin de récupérer son état. | **FabriqueImage** : C’est la fabrique d’image qui utilise la classe Image et aucune autre classe. Dès lors, Image dépend de la classe fabrique par application du patron de conception « Fabrique ». |
| **ImageSnap** | **ImageSnap (double zoom, Int posX, Int posY**) : Constructeur par défaut de la classe ImageSnap, qui va permettre de créer une sauvegarde de l’image par copie de sa position à l’écran mais aussi son zoom. | **FabriqueImage** : Idem que la classe Image, c’est la fabrique d’image qui instancie une nouvelle ImageSnap. |
| **Gardien** | **Application du patron Mémento**  **ajouterSnap (Perspective state**) : Méthode qui permet de sauvegarder une perspective dans le mémento.  **restorLast ()** : Méthode qui permet de retourner la dernière perspective de l’image.  **restorNext (**) : Méthode qui permet de retourner une plus ancienne perspective de l’image. | **Perspective** : C’est la perspective qui dispose d’un gardien afin de pouvoir en sauvegarder les différentes modifications subies. Ainsi, le gardien dépend de la classe Perspective. |
| **Originator** | **Application du patron Mémento**    **SetSave (Perspective maperspective)** : Méthode set Save va permettre de récupérer les données comprise dans une perspective pour ensuite les stocker dans le mémento.  **CreateSave ()** : Méthode qui permet de retourner une instance de la ImageSnap par utilisation de la farbriqueImage. Ainsi, CreateSave () reçoit en paramètre les positions X et Y de l’image sur le panneau d’Image ainsi que son zoom pour en faire une mémorisation  **restoreFromMemento (ImageSnap snap)** : Méthode qui permet de retourner une Perspective selon une sauvegarde reçu en paramètre pour pourvoir la ré-appliquer. | **Gardien** : C’est la classe gardien qui dispose d’un Originator pour avoir la possibilité de sauvegarder les changements d’état de la perspective. |
| **Perspective** | **Application du patron Mémento**  **incrementeZoom ()** : Méthode qui permet de d’augmenter le zoom d’une perspective.  **decrementeZoom ()** : Méthode qui permet de diminuer le zoom d’une perspective.  **setPositionX (Int deplacementRecuX)** : Méthode qui va permettre de définir la position x de l’image à l’écran.  **setPositionY (Int deplacementRecuY)** : Méthode qui va permettre de définir la position Y de l’image à l’écran.  **restorePerspective (Perspective snap)** : Méthode qui permet de restaurer une perspective sur une Image. Cette méthode reçoit en paramètre une ancienne valeur de perspective pour ensuite la ré-appliquer  **save ()** : Cette méthode permet de mémoriser dans le mémento une l’état de la perspective à l’instant T.  **collerUnePerspective (Int [] tab)** : Cette méthode permet de copier une valeur mémorisée par le clic droit > copier perspective afin de la ré-appliquer sur la perspective en question.  **clone ()** : Méthode qui permet de créer une copier de la perspective actuelle. | La perspective dépend de toutes les c**ommandes** mais aussi de l**’image**.  Les **commandes** vont être appliquées des modifications sur la perspective et l’**image** dispose d’une perspective. |
| **Observer** | **Application du patron de conception Observer**  **Observer = PanneauImage.**  **Update ()** : Méthode qui permet de réactualiser le panneau d’image dès l’avertissement de la Perspective d’une modification des données. | Observable.  Ainsi, La classe PanneauImage dépend de la classe Perspective (la classe observée. |
| **Observable** | **Application du patron de conception Observer**  **Observable = Perspective.**  **Notify ()** : Méthode qui permet d’avertir les observateur d’une modification des données dans la Perspective afin qu’ils puissent se mettre à jour. |  |
| **Vue** |  |  |
| **PanneauImage** | **Paint (Graphics g)** : Méthode qui permet de dessiner l’image sur le panneau d’image.  **setImage (String lienImage)** : Méthode qui permet de faire appel à la fabrique d’image en recevant le lien de l’image à afficher.  **reactualiserPanneauImage ()** : Méthode qui permet de réactualiser le panneau d’image.  **Update ()** : Méthode qui sera déclenchée dès que la Perspective de l’image sera modifiée. Dès lors, update () fera appel à la méthode reactualiserPanneauImage (). | **FenetrePrincipale** : C’est la fenêtre principale qui instancie et dispose du panneau d’image |
| **FenetrePrincipale** | **FenetrePrincipale ()** : Constructeur par défaut de la classe **FenetrePrincipale** qui va instancier la Frame de l’application, Le menu ainsi que le panneau d’Image. | **DemarrerApplicationGraphique :** C’est à l’aide de cette classe qu’on utilise le constructeur de la FenetrePrincipale. |
| **Contrôleur** |  |  |
| **EcouteurEvenements** | **keyReleased (KeyEvent arg0)** : méthode qui va permettre de savoir si l’utilisateur à fait l’action CTRL Z ou Y afin de défaire ou refaire un état sur la perspective. Dès lors, cela implique l’utilisation du singleton et l’utilisation des commandes CTRLY et CTRLZ  **mousePressed (MouseEvent arg0)** : Méthode qui va permettre de savoir si l’utilisateur a cliqué sur le bouton gauche ou droit de la souris. Si l’utilisateur clique sur le bouton gauche on retient ses coordonnées, sinon on déclenche le menu pop-up (clic droit)  **mouseWheelMoved (MouseWheelEvent arg0)** : Méthode qui permet de savoir si l’utilisateur a fait une action avec la molette de la souris. Ainsi, cela implique soit un zoom soit de dé-zoomer la perspective par le biais du singleton qui appliquera la commande adéquate.  **mouseDragged (MouseEvent e)** : Méthode qui permettra de savoir si l’utilisateur déplace la souris, ce qui implique une translation par le biais du singleton. | **PanneauImage :** Ainsi, c’est sur le panneau d’image que s’appliquent les écouteurs d’événements. Donc, les écouteurs dépendent du panneau d’image. |
| **Menu** | **addMenuFichier ()** : cette méthode dispose d’un écouteur d’évènement associé à la sauvegarde du fichier image, mais aussi pour la création d’une image.  **addMenuModifier ()** : cette méthode dispose d’un écouteur au niveau de son menu modifier. Dès lors, si l’utilisateur clique sur CTRL Z OU Y cela implique soit un retour à l’état suivant ou bien de revenir à l’état initial avant le CTRL Z. Ainsi, on utilise le singletonCommande pour appliquer la bonne commande selon l’évènement. | **FenetrePrincipale :**  C’est sur la FenetrePrincipale que l’on applique le menu. Donc, le Menu dépend de la fenêtre principale. |
| **PopMenuClicDroit** | Cette classe dispose d’un écouteur d’événements permettant de savoir si l’utilisateur a cliqué sur un item du pop-up menu.  Si on clique sur copier perspective : on retient l’état de la perspective.  Si on clique sur Coller perspective : on utilise le singletonCommande afin d’appliquer la commande Coller.    **déclanchement (Component comp, Int x, Int y)** : Méthode qui déclenche le menu pop-up clic droit. | **Panneau-Image et de l’écouteur d’événements.** |
| **SingletonCommande** | **Application du patron Singleton**  **getInstance ()** : Méthode qui permet de retourner une instance du singleton (instance unique).  **Exécution ()** : méthode qui permettra d’exécuter la bonne commande selon les valeurs reçues des inspecteurs d’événements. | **Menu, PopMenuClicDroit, EcouteurEvenements :** ce sont ces trois Classes qui utilisent l’instance de SingletonCommandes afin de traduire les interactions utilisateur en commandes à appliquer. |
| **« Abstract »**    **Commande** | **Application du patron Commande**  **Exécution (Panneau-Image image)** : les classes qui hériteront de la classe Commandes devront implémenter leur propre méthode d’exécution. | **SingletonCommandes** :  C’est le SingletonCommandes qui instanciera les commandes à appliquer selon les évènements reçus des différents intermédiaires d’événements. |
| **Coller** | **Application du patron Commande**  **Exécution (Panneau-Image image)** : Méthode qui va permettre d’indiquer à la perspective de modifier ses valeurs par celles d’une perspective qui aura été copiée au préalable. (reçues en paramètres) | **SingletonCommandes** :  Idem que la classe Commande |
| **Sauvegarde-Perspective** | **Application du patron Commande**  **Exécution (Panneau-Image image)** : Méthode qui va permettre d’indiquer à la perspective de sauvegarder ses valeurs dans le mémento. | **SingletonCommandes** :  Idem que la classe Commande |
| **CtrlZ** | **Application du patron Commande**  **Exécution (Panneau-Image image)** : Méthode qui permettra d’indiquer à la perspective de restaurer son état à celui de l’instant T-1. | **SingletonCommandes** :  Idem que la classe Commande |
| **CtrlY** | **Application du patron Commande**  **Exécution (Panneau-Image image)** : Méthode qui permettra d’indiquer à la perspective de restaurer son état à celui de l’instant T+1. | **SingletonCommandes** :  Idem que la classe Commande |
| **Serializer** | **Application du patron Commande**  **Exécution (Panneau-Image image)** : Méthode qui permettra d’indiquer à l’image de sauvegarder son état sur disque local. | **SingletonCommandes** :  Idem que la classe Commande |
| **Deserializateur** | **Application du patron Commande**  **Exécution (PanneauImage image)** : Méthode qui permettra de reconstruire une image et sa perspective attachée grâce à une sauvegarde faite auparavant | **SingletonCommandes** :  Idem que la classe Commande |
| **Translation** | **Application du patron Commande**  **Exécution (PanneauImage image)** : Méthode qui va permettre d’indiquer à la perspective de modifier ses positions sur l’écran en changeant ses valeurs X et Y par celles reçues en paramètre. | **SingletonCommandes** :  Idem que la classe Commande |
| **Zoom** | **Application du patron Commande**    **Exécution (PanneauImage image)** : Méthode qui permettra d’indiquer à la perspective de modifier son zoom soit en l’incrémentant ou en le décrémentant selon la molette de la souris. | **SingletonCommandes** :  Idem que la classe Commande |
| **Lanceur** |  |  |
| **DemarrerApplicationGraphique** | Permet de lancer l’application |  |

## 2.2 Architecture logicielle

2.3. Description de l'architecture

**Ce diagramme de classe UML a été élaboré avec le logiciel Star UML.**

**MODÈLE**

**Classe Image :**

La classe Image possède comme attribut une image concrète, le chemin de l’image concrète et une perspective. Elle implémente Serializable, car elle fait partie de la sauvegarde. Par contre, l’image concrète, elle ne fait pas partie de la sauvegarde, elle est donc déclarée comme « transient ». Les instances de cette classe correspondent à l’image affichée dans les panneaux image.

**Classe FabriqueImage :**

Cette classe s’occupe simplement de créer les instances de la classe Image et ImageSnap. C’est l’application du patron Fabrique. Les deux méthodes de cette classe renvoient des instances et sont déclarées statiques pour que ces méthodes soient accessibles partout. Ces méthodes sont appelées seulement dans la classe PanneauImage, d’où leur relation dans le diagramme UML.

**Classe ImageSnap :**

ImageSnap correspond à la classe qui sauvegarde les perspectives (position x et y et sa valeur de zoom) de l’image tout au long de ses modifications comme des translations ou des zooms. La classe Gardien possède deux piles d’ImageSnap qui correspond aux modifications antérieures et postérieures. Les instances de cette classe sont créées par la classe FabriqueImage par le biais de la classe Originator qui s’occupe des sauvegardes.

**Classe Originator :**

Cette classe gère les sauvegardes d’Image selon le patron Memento. Elle a des liens avec ImagesSnap, car elle s’occupe des sauvegardes des instances de celles-ci et permet de restaurer des ImagesSnap à partir d’une instance d’ImageSnap mise en sauvegarde.

**Classe Gardien :**

Cette classe contient deux piles d’ImageSnap pour sauvegarder les modifications affectées à l’image. Il contient aussi une instance d’Originator, car c’est lui qui l’instancie. Cette classe est utilisée pour le patron Memento. C’est lui qui contient la liste de sauvegardes (les deux piles).

**Classe Perspective :**

Perspective correspond aux caractéristiques de l’image (son hauteur, sa largeur, sa position, etc). Elle est l’une des classes centrales de l’application. Cette classe hérite de Observable ce qui permet de notifier les observateur lors d’une modification de la perspective. La classe implémente Serializable, car elle est contenue dans la sérialisation d’une image. De plus, une instance de Perspective est attachée à chaque Image comme dit auparavant.

**Classe NullPerspective**

Cette classe définit un objet Perspective vide de la classe Perspective selon le patron Null Object. Cette classe hérite donc de perspective et permet d’éviter de retourner null et d’avoir des comportements non gérés.

**Classe Observer**

Cette classe définit le comportement des observateurs en les obligeant à redéfinir la méthode update appelée lors d’un notify de la classe observée. La classe PanneauImage est une classe qui implémente Observer.

**Classe Observable**

Cette classe permet de créer une sorte d’écouteur sur une classe pour plusieurs autres classes selon le patron Oservateur. Elle contient une méthode pour ajouter un observateur et une autre pour notifier les observateurs.

**VUE**

**Classe FenetrePrincipale :**

Cette correspond à la fenêtre de l’application. C’est elle qui contient les différents panneaux image et le menu.

**Classe PanneauImage :**

Cette classe correspond aux panneauImage qui contiennent chacun une image. C’est elle qui s’occupe de toutes les fonctions correspondant à l’image comme la redessiner, la changer. Elle hérite donc de JPanel et implémente Observer pour recevoir les notifications des classes écoutées.

**CONTROLEUR**

**Classe EcouteurEvenement :**

Cette classe implémente quasiment toutes les « interfaces d’actions » d’une souris (MouseListener, MouseWheelListener,KeyListener,MouseMotionListener). Cela permet donc d’effectuer une méthode correspondant à chaque action de la souris. Cette classe contient en attribut un objet PanneauImage à écouter, un menu disponible avec le clic droit et différentes variables.

**Classe Menu :**

La classe Menu correspond au menu dans la fenêtre principale. L’instance actuelle de la fenêtre principale est récupérée lors de la création du menu pour effectuer des actions dessus.

**Classe PopMenuClicDroit :**

Cette classe correspond au menu qui apparait lors d’un clic droit sur un PanneauImage. Une instance de cette classe est contenue dans la classe EcouteurEvenement. Lors de la création d’une instance de ce menu le panneauImage actuel est récupéré.

**Classe SingletonCommande :**

Singleton vérifie et contrôle qu’il existe bien qu’une instance de cette classe à tout moment selon le patron de conception « Singleton ». C’est cette classe qui effectue toutes les actions à faire sur les images contenues dans les différents panneaux, en créant une commande puis en l’exécutant selon le patron de conception « Commande ».

**Classe Commande :**

Commande est la classe abstraite qui définit les méthodes d’exécution des différentes commandes possibles sur une image. Elle est donc abstraite, il est impossible de l’implémenter et redéfinie la méthode exécution qui sera implémentée dans chaque classe « action » comme translation, zoom, etc.

**Classe CtrlY :**

Cette classe définie le comportement lors d’un appui sur les touches CTRL+ Y de la part de l’utilisateur. Elle implémente commande selon le patron de conception « Commande » et redéfinie donc la méthode exécution.

**Classe CtrlZ :**

Cette classe définit le comportement lors d’un appui sur les touches CTRL+ Z de la part de l’utilisateur. Elle implémente commande selon le patron de conception « Commande » et redéfinie donc la méthode exécution.

**Classe Serializateur :**

Cette classe définit le comportement lors d’une demande de sauvegarde de la perspective actuelle de l’image de la part de l’utilisateur. Elle implémente commande selon le patron de conception « Commande » et redéfinie donc la méthode exécution.

**Classe DeSerializateur :**

Cette classe définit le comportement lors d’une demande d’ouverture de sauvegarde d’une image de la part de l’utilisateur. Elle implémente commande selon le patron de conception « Commande » et redéfinie donc la méthode exécution.

**Classe Translation :**

Cette classe définit le comportement lors d’une translation sur l’image de la part de l’utilisateur. Elle implémente commande selon le patron de conception « Commande » et redéfinie donc la méthode exécution.

**Classe Zoom :**

Cette classe définit le comportement lors d’un zoom sur l’image de la part de l’utilisateur. Elle implémente commande selon le patron de conception « Commande » et redéfinie donc la méthode exécution.

**Classe Dessiner :**

Cette classe définit le comportement lors d’un dessin (avec la molette) sur l’image de la part de l’utilisateur. Elle implémente commande selon le patron de conception « Commande » et redéfinie donc la méthode exécution.

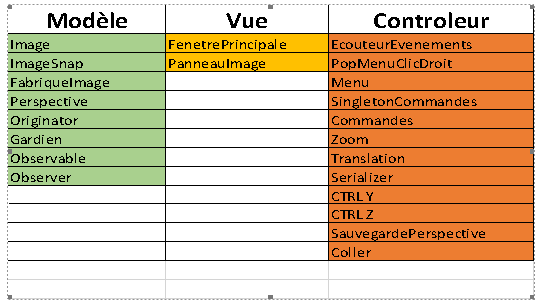
**Classe Coller :**

Cette classe définit le comportement lors d’un « copier » de l’utilisateur, cela va copier en mémoire la perspective actuelle. Elle implémente commande selon le patron de conception « Commande » et redéfinie donc la méthode exécution.

## 2.4. Utilisation des patrons de conception

### 2.4.1. PATRON « Modèle Vue Contrôleur »

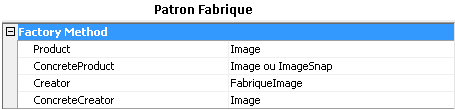
Voici ci-dessous le tableau expliquant la correspondance entre l’application du patron MVC et les éléments de notre structure. Ce tableau résume les classes qui sont incorporées soit dans le modèle, la vue ou dans le contrôleur.



Nous avons appliqué ce patron de conception car cela permet d’obtenir un avantage consécutif dans l’architecture de notre application. Ainsi, appliquer une telle architecture au logiciel engendre une certaine clarté dans le fonctionnement de notre application. Grâce au MVC, cela simplifie la tâche du développeur qui tenterait d'effectuer une maintenance ou une amélioration sur le projet. De ce fait, la modification des traitements de l’application ne change en rien la vue. Par contre, cela implique que chaque vue devra disposer de son propre contrôleur afin de pouvoir traduire les événements utilisateurs en en requêtes pour que le modèle puisse mettre à jour ses données.

**2.4.2. Patron « FABRIQUE»**

Voici ci-dessous le tableau expliquant la correspondance entre l’application du patron Fabrique et les éléments de notre structure.



L’application de ce patron consiste à déléguer l'instanciation d'objets à une classe. Ceci implique que la classe source de l'objet n'est donc pas connue par le client. Pour notre application, cela est une base importante qui nous permet d’instancier des images par le Panneau-Image tout en passant par la fabrique afin que le Panneau-Image ne connaisse pas la classe Image. C’est une sorte d’application de la loi de Déméter.

### 2.4.3. PATRON « SINGLETON»

Voici ci-dessous le tableau expliquant la correspondance entre l’application du patron Singleton et l’élément de notre structure.

### 

L’utilisation de ce patron permet d’avoir une instance unique d’une classe. Dès lors, l’accès à cette instance est contrôlé par la classe elle-même et plusieurs classes peuvent utiliser l’instance unique de la classe en question. Dans le cas de notre application, notre SingletonCommande permet de restreindre l'instanciation de la classe à un seul objet. De ce fait, les commandes adéquates pourront être appliquées depuis le SingletonCommande par simple récupération de l’instance du singleton en lui transmettant les paramètres nécessaires.

### 

### 2.4.4. PATRON « COMMANDE »

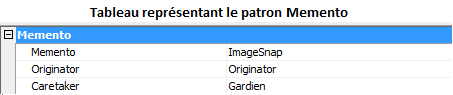
Voici ci-dessous le tableau expliquant la correspondance entre l’application du patron Commande et les éléments de notre structure.

### 

L’application de ce patron nous permet de découpler l’objet qui invoque la commande de celui qui la recevra. Ainsi, cela nous permet de respecter l’application du patron de conception MVC. Dès que l’utilisateur fera une action qui sera repérée par le « receiver » soit l’EcouteurEvenement ou le Menu de notre application, ils avertiront le contrôleur afin d’exécuter la commande adéquate. Par conséquent, la vue qui est à l’origine de la détection de l’évènement ne sera pas en liaison directe avec la classe Perspective. De plus, si modification il y a, c’est le SingletonCommande et la Commande adéquate qui sera à l’origine de la modification.

### 2.4.5. PATRON « MÉMENTO»

Voici ci-dessous le tableau expliquant la correspondance entre l’application du patron Mémento et l’élément de notre structure.



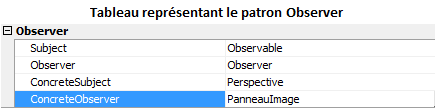
L’application du patron de conception mémento permet de faire des sauvegardes de l’état, voir d’une partie de l’ état d’un objet afin de le retourner à l’utilisateur en cas de besoin. Dans notre cas, le patron mémento nous permet de sauvegarder l’état de notre image, soit de la perspective, afin de pouvoir défaire ou refaire des modifications sur la perspective. Ainsi, nous pouvons utiliser les commandes CTRL-Z et CTRL-Y sur notre perspective.

### 

### 

### 2.4.6. PATRON « OBSERVER»

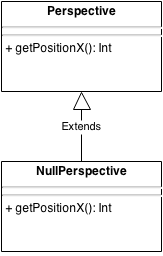
Voici ci-dessous le tableau expliquant la correspondance entre l’application du patron Observer et l’élément de notre structure.



L’application du patron observer permet à la classe Observable de notre application d’envoyer des signaux à une autre classe Observatrice afin de réactualiser ses données. Dans notre cas, la classe observable est la classe Perspective et l’observateur est le PanneauImage. Dès que la perspective sera modifiée, celle-ci enverra un avertissement au panneau d’image afin qu’il effectue alors l'action adéquate, soit une réactualisation des données.

**2.4.7. PATRON « Null Object»**

**NullObject > Null Perspective**



L’application du patron “Null Object” permet de ne pas retourner de null à la place d’une perspective.

La classe “NullPerspective” hérite de la classe Perspective, afin de pouvoir utiliser le polymorphisme lors des retours des méthodes sensées retourner une Perspective.

Toutes les méthodes de la classe Perspective sont surchargées dans la classe NullPerspective, afin de retirer le comportement.

L’utilisation de ce design pattern permet de ne pas avoir d’erreur lors de l’exécution du programme au cas où une méthode ne pourrait pas retourner une perspective.

**2.4.8. PATRON « Immutable»**

L’application du patron “Null Object” permet de bloquer la modification d’un attribut après sa création, ceci permet d’éviter qu’un attribut qui ne doit pas être modifié le soit par erreur et place le programme dans un état incohérent.

Dans le cas du laboratoire, l’attribut contenant le chemin de l’image sur le disque dur n’a pas besoin d’être modifié après la création de la classe Image.

## 2.4.Faiblesses de la conception

**2.4.1) Sérialisation du BufferedImage**

Après analyse de notre application, nous avons décelé une faiblesse au niveau de la sauvegarde de l'état de nos perspectives sur le disque du client. Ne pouvant pas sérialiser l’attribut “BufferedImage”, soit l’image concrète du Modèle.Image, cela nous impose donc de disposer de l’image originale sur le disque pour pouvoir ouvrir de nouveau le fichier. En sommes, nous devons passer par le chemin de l’image sur le disque pour la redessiner. En conséquent, si l’utilisateur ne dispose plus de l’image source après une sauvegarde, il ne pourra plus l’utiliser et ainsi la réafficher. Pour remédier à ce problème, il faudrait intégrer l’image d’origine dans la sérialisation afin de pouvoir l’utiliser même si l’image d’origine n’est plus présente sur le disque dur du client.

**2.4.2) Amélioration de la sauvegarde**

Nous avons aussi décelé une faiblesse au niveau de l’exécution de la sauvegarde des états de nos perspectives. Pour sauvegarder l’état de notre Modèle visible dans la vue, nous devons actuellement pointer le Menu de la fenêtre puis fichier > Sauvegarder. Cette action implique l’appel d’une instance de “SingletonCommande” avec les paramètres suivants :

SingletonCommande.execution(8, new int[1],fenetrePrincipale.panneauImage).

(8 : correspond au numéro de la commande, int lambda, le panneau Image)

Nous pouvons voir que lorsque nous voulons sauvegarde, nous envoyons en paramètre seulement le panneau1 (Vue1) de la fenêtre principale. Ceci est une lacune due à la commande abstraite exécution () du “SingletonCommande” qui exige les trois paramètres cités ci-dessus pour s’implémenter et s’exécuter. Par conséquent, dans le “SingletonCommande” nous devons chercher l’attribut panneau2 de la fenêtre principale directement depuis l’instance de la fenêtre principale de l’application, afin de pouvoir sauvegarder l’ensemble des états des modèles. Il serait donc préférable dans une évolution future de l’application de revoir le mode de fonctionnement de la commande sauvegarde pour en améliorer son efficacité et sa sécurité. Cette commande devrait donc recevoir directement en paramètre tous les modèles à sauvegarder dans le fichier depuis l’instance du “SingletonCommande”.

**2.4.3) Amélioration du MVC**

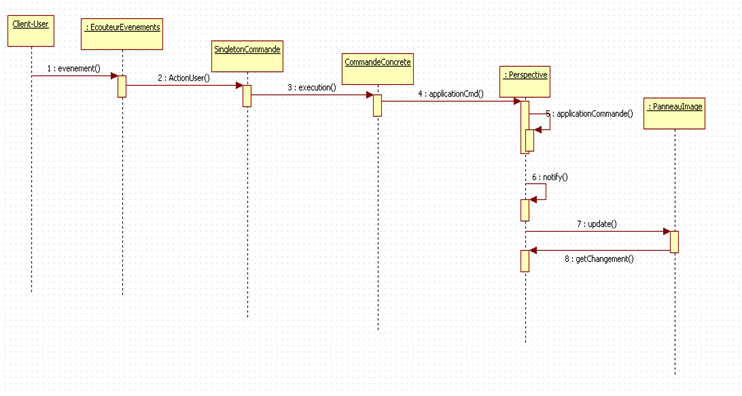
Après réalisation de notre application, il nous est apparu une faiblesse concernant l’ouverture des fichiers de sauvegarde “.psg”. En effet, dès que l’on ouvre un fichier d’extension “.psg” cela implique l’utilisation de la commande désérialisation depuis le “SingletonCommande” qui permet de récupérer l’ensemble des données de la sauvegarde. ( es trois perspectives et les liens de l’image). Dès lors, la mise à jour des panneaux de la vue se fait depuis le Menu de l’application. Ceci est en effet une faiblesse car cela va à l’encontre du patron MVC. Pour résoudre cette faiblesse, il aurait fallu mettre en place une commande permettant de mettre à jour les modèles et ainsi déclencher une notification aux différents panneaux pour qu’ils puissent en récupérer les valeurs.

**2.4.4) Amélioration de l’Observateur**

A la suite de la réalisation, on s'est aperçu que notre commande Notify() est certes fonctionnelle, mais n’indique pas qu’elle est la modification apportée sur le modèle. Dès lors, nous sommes obligés de mettre à jour toutes les données du modèle sur la vue. Ceci peut entraîner à longue un engorgement dans les flux de l’application. La solution que l’on aurait pu apporter à ce problème aurait été d’ajouter un attribut de type Objet dans le Notify() de la classe observée afin d’envoyer à la vue (par update() de l’observateur) directement l’élément à mettre à jour dans sa vue pour le client.

## 2.5.Diagramme de séquence (uml)

**Dynamique de l'architecture MVC : Générale**



Le diagramme de séquence ci-dessus nous permet de voir l’application du patron de conception. Par conséquent, dès que le client fera une modification sur la vue, l’écouteur d’événement situé dans le contrôleur traduira ceci en action, en appelant le singleton de commande pour qu’il puisse appliquer la commande adéquate. Dès lors, une modification s'établira sur la perspective. Une fois la modification établie, celle-ci envoie une notification au panneau d’image afin qu’il puisse se mettre à jour.

### 

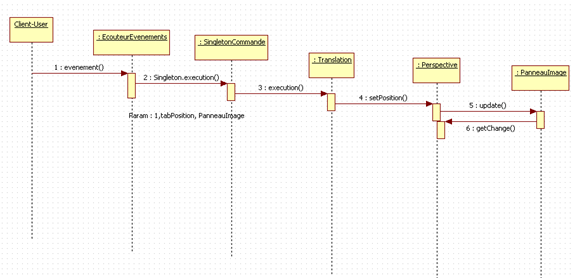
### 

### 

### 

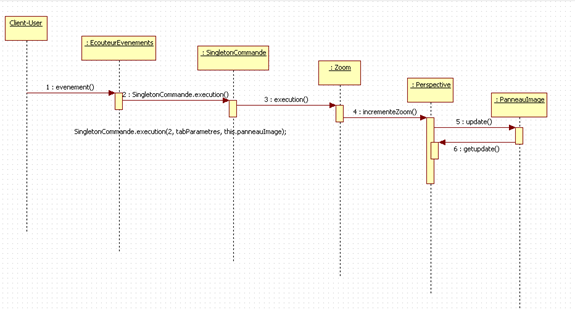
### 2.5.1. Dynamique de l'architecture MVC : effectuer une certaine commande

### Diagramme de séquence sur l’application d’une commande Translation :



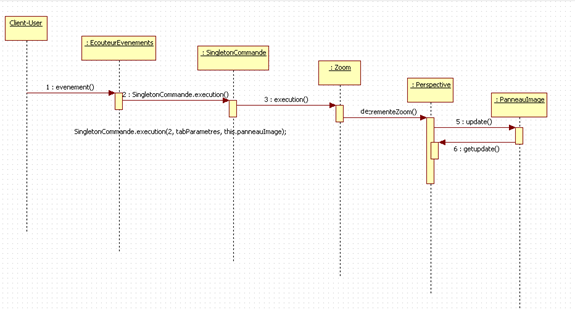
Avec ce diagramme de séquences, nous pouvons voir comment se déroule la commande translation. Dès que le client faira une modification sur la vue avec la souris, l’écouteur d’événement situé dans le contrôleur traduira ceci en action en appelant le singleton de commande pour qu’il puisse appliquer la commande translation. Dès lors, la commande translation va appliquée sur la perspective la commande setPosition() (avec la récupération de la position de la souris dans l’écouteur d’événement)pour modifier la position de l’image à l’écran. Une fois la modification établie, celle-ci envoie une notification au panneau d’image afin qu’il puisse se mettre à jour.

### Diagramme de séquence sur l’application d’une commande Zoom Avant :



Avec ce diagramme de séquences, nous pouvons voir comment se déroule la commande Zoom avant. Dès que le client fera une modification sur la vue avec la molette de la souris, l’écouteur d’événements situé dans le contrôleur traduira ceci en action en appelant le singleton de commande pour qu’il puisse appliquer la commande zoom. Si la commande Zoom reçoit un 1 comme paramètre cela équivaut à un zoom avant. Dès lors, la commande appliquera sur la perspective la commande incrémenteZoom() et cela modifiera le zoom de l’image en l’agrandissant. Une fois la modification établie, celle-ci envoie une notification au panneau d’image afin qu’il puisse se mettre à jour.

### Diagramme de séquence sur l’application d’une commande Zoom Arrière :



Avec ce diagramme de séquences, nous pouvons voir comment se déroule la commande Zoom arrière. Dès que le client fera une modification sur la vue avec la molette de la souris, l’écouteur d’événements situé dans le contrôleur traduira ceci en action en appelant le singleton de commande pour qu’il puisse appliquer la commande zoom. Si la commande Zoom reçoit un 0 comme paramètre cela équivaut à un zoom arrière. Dès lors, la commande appliquera sur la perspective la commande decrémenteZoom() et cela modifiera le zoom de l’image en le réduisant. Une fois la modification établie, celle-ci envoie une notification au panneau d’image afin qu’il puisse se mettre à jour.

### 

### Diagramme de séquence sur l’application d’une commande Serializer :

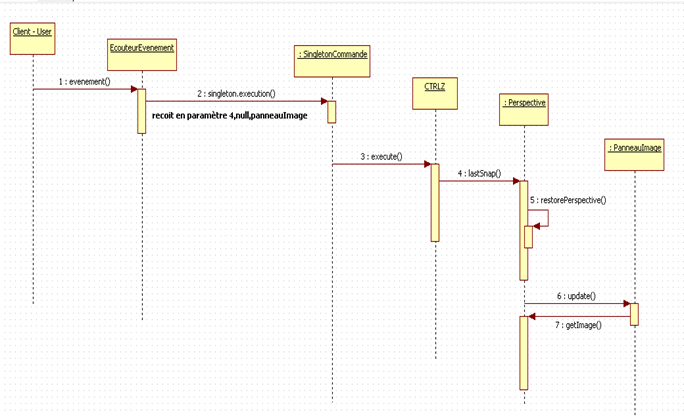
### 

### Avec ce diagramme de séquences, nous pouvons voir comment se déroule la commande qui permet de sauvegarder sur le disque local l’état actuelle de notre objet. Dès que le client accédera au menu Fichier > Sauvegarder, l’écouteur d’événement du menu situé dans le contrôleur traduira ceci en action en appelant le singleton de commande pour qu’il puisse appliquer la commande serializer. La commande serializer reçois en paramètre l’image actuelle à sauvegarder. Dès lors, la commande appliquera sa sauvegarde sur le disque grâce au objets “FileOuputStream” et “ObjectOutputStream”. Le fichier portera comme extension “.psg” et se localisera “répertoire source du projet\src\image”.

### 

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

**Diagramme de séquence sur l’application d’un retour sur commande Lambda depuis MVC :**



### Avec ce diagramme de séquences, nous pouvons voir comment se déroule le processus qui permet de défaire une commande. Dès que le client fera une modification sur la vue avec le clavier ou le menu, l’écouteur d’événement situé dans le contrôleur traduira ceci en action en appelant le singleton de commande pour qu’il puisse appliquer la commande CTRL-Z ou CTRL-Y. Dès lors, la commande appliquera sur la perspective la commande last() et cela permettra de modifier la perspective à un état antérieur grâce au gardien donc au patron mémento. Une fois la modification établie, celle-ci envoie une notification au panneau d’image afin qu’il puisse se mettre à jour.

# 

# 

# 

# **3.Décisions d'implémentation**

## 3.1.Décision 1 : Choix sur l’implémentation du patron Observer Observable

Dans le cadre de ce laboratoire, nous voulions absolument faire en sorte que notre application soit en adéquation avec le patron de conception Modèle Vue Contrôleur.

**Choix 1** : Nous avions pour choix numéro un de faire directement une liaison entre la classe “Panneau-image” et “Perspective” afin qu’ils puissent communiquer directement sans intermédiaire.

**Choix 2** : Nous avions pour deuxième option de mettre en place le patron de conception Observer Observable et définir en tant que classe observable la Perspective le panneau d’image en Observer.

Dans le cadre de notre application, nous utilisons le patron de conception Modèle Vue Contrôleur permettant de rendre flexible et facilement modulable notre structure en cas de modifications futures. Ainsi, nous avons décidé d’implémenter la solution numéro 2 Observer Observable. En intégrant ce patron de conception cela nous permet d’abonner nos vues au modèle (Perspective). Par conséquent, notre panneau d’image doit s’abonner au sujet auquel il porte son intérêt. En sommes, dès que le modèle se met à jour il avertit les vues par une méthode Notify() qu’un événement s’est passé et qu’une mise à jour est nécessaire, par la méthode update() de l’abonné.

**3.2.Décision 2 : Choix sur l’implémentation d’un écouteur d’événement unique (incluant MouseListener, MouseWheelListener,KeyListener, MouseMotionListener )**

Dans le cadre de ce laboratoire, il nous est impératif de mettre en place des écouteurs permettant de traduire les actions faites depuis les différentes Vue de l’application par le client. Mettre en place des écouteurs d’objets nous permettant de comprendre ce que l’utilisateur a tenté de réaliser afin de faire les modifications ou d'aviser une autre classe qui se chargera de la modification sur le Modèle.

**Choix 1** : Nous avions pour choix numéro un de faire différents écouteurs d’événements afin de traduire les actions utilisateurs en commande à exécuter sur le Modèle.

**Choix 2** : Nous avions pour deuxième option de mettre une classe globale regroupant l’ensemble des interfaces permettant d’intercepter les actions de l’utilisateur afin de les traduire en action.

Après analyse des deux choix, nous avons décidé de valider la deuxième solution mettant en évidence la mise en place d’une classe unique implémentant les différents écouteurs. Cette pratique nous a permis de simplifier notre conception d’application, telle une généralisation de cette même classe nommée “EcouteurEvenement” servira de “listener” pour chaque Vue. Le rôle de cette classe sera donc de fournir une description et un décodage de l’événement capturé afin de pouvoir indiquer au “SingletonCommande” la bonne commande à effectuer.

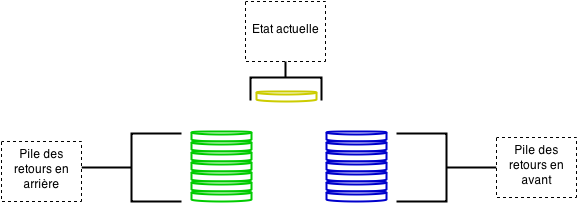
## 3.3.Décision 3 : Utilisation des piles de sauvegarde des transitions de la perspective.

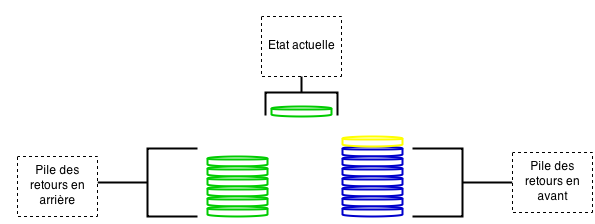
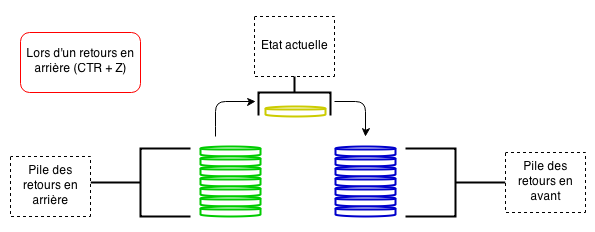
Pour la gestion des retours en arrières et des retours en avant (CTRL + Z et CTRL + Y) il fallait pouvoir se déplacer dans les mémentos des perspectives vers l'avant ou l'arrière. Pour cela, il a fallu réfléchir à un moyen de stocker les mémentos de manière à pouvoir se déplacer facilement.

2 solutions ont été pensées pour résoudre ce problème :

**Choix 1** : La première consistait à 'utiliser un système comprenant 2 piles (ou Stack), une qui comprend les retours en arrières possibles, et une pour les retours en avant possibles.

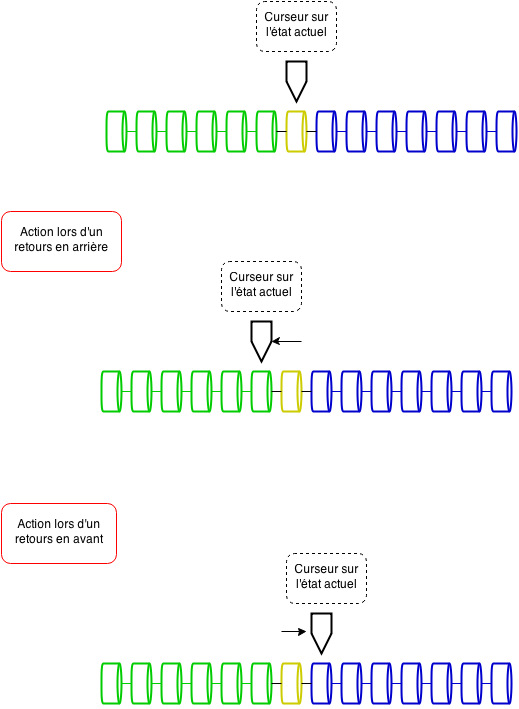
Lorsqu'un retour en arrière est effectué, la sauvegarde actuelle est placée dans la pile des retours en avant possibles, et vice-versa. Cette méthode permet de séparer les retours en avant des retours en arrière, afin d'être sûr de ne pas se tromper lors de la navigation dans les mémentos





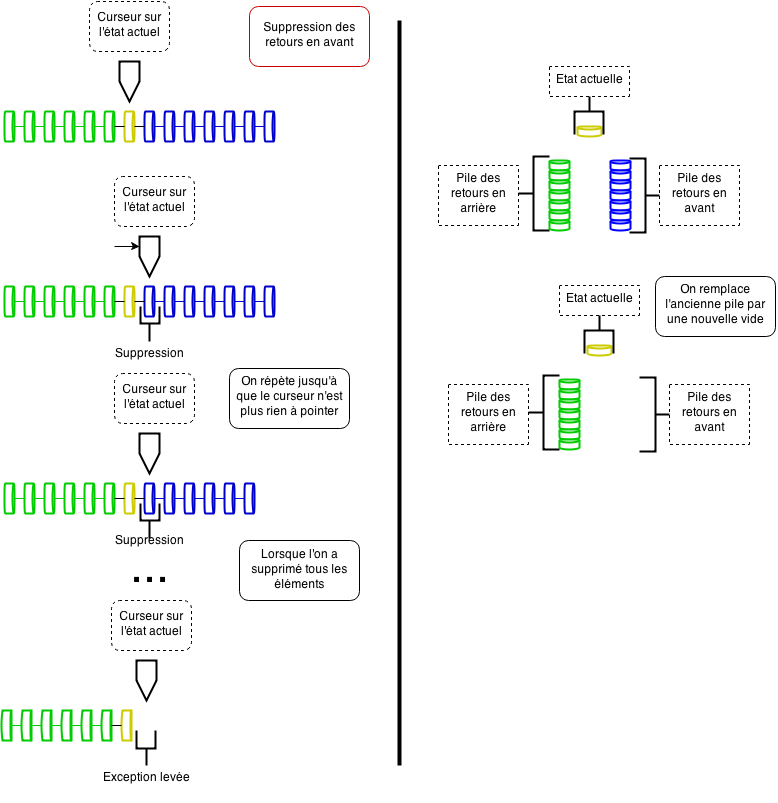
*Schéma représentant l’utilisation de la pile lors d’un retours en arrière*

**Choix 2** : La 2ème solution était de stocker tous les mémentos dans un vecteur (Vector), et d'utiliser un itérateur pour se déplacer dedans. Cette itérateur garderait en mémoire la position actuelle dans la liste des memento, et se déplacerait vers l'avant pour un CTR + Y, et en arrière pour un CTR + Z



*Schéma représentant l’utilisation d’un vecteur pour la gestion des memento*

Il a été choisi d'utiliser la première solution, celle-ci permet de facilement se déplacer dans les mémentos sans avoir à se soucier de la position actuelle, les mémentos était empilés lors de la sauvegarde, et dépilés lors du retour en avant ou en arrière. De plus, il est facile de remettre à zéro la pile des CTR + Y lors d'une nouvelle action de l'utilisateur. Avec l'utilisation d'un vecteur, il aurait fallu se déplacer de la position de l'itérateur actuelle jusqu'à la fin de la liste pour en supprimer tous les éléments pour remettre à zéro.



*Schéma représentant le comportement d’un vecteur et d’une pile lors de la suppression des retours en avant*

## 

## 3.4.Décision 4 : Choix d’implémentation sur la sérialisation et la désérialisation

Au cours de ce laboratoire, nous avons dû nous pencher sur la solution permettant d’établir la sauvegarde de notre état global sur le disque.

**Choix 1** : En choix numéro un : sauvegarder l’image concrète (BufferedImage) actuelle dans le fichier de sauvegarde

**Choix 2** : En choix numéro deux : déclarer l’attribut “BufferedImage” du Modèle.Image en transient pour ne pas l’inclure dans la sérialisation de notre état.

Nous avons choisi la solution deux qui permet de rendre plus légère la sauvegarde et plus maniable. Aussi, il est plus compliqué de sérialiser une classe java car son fonctionnement n’est pas définie par nous même donc il peut y avoir des ambiguïtés. Par conséquent, c’est pour cela que nous avons choisi la solution 2. Pour sauvegarder nous devons donc appliquer une commande nommée sérializer(sous-classe de Commande) qui nous permet de sauvegarder les états de nos trois perspectives selon nos vues dans un fichier d’extension “.psg”. En sommes, ce nouveau fichier contient toutes les données de nos trois panneaux afin de restaurer leur états dans une réouverture future.

Pour ouvrir un fichier d’extension “.psg”, nous devons obligatoirement passer par une étape de désérialisation de notre fichier. Le principe de la désérialisation est de pouvoir extraire les données contenues dans notre fichier selon une certaine extension. Ainsi, par le biais de la classe Désérializer (sous-classe de Commande) dès la réception d’un ordre de commande, celle-ci extraira nos trois objets de type Modèle.Image afin d’en retourner un tableau de Modele.Image pour qu’ils puissent être réaffectés aux différentes vues de notre application. Ce qui nous permet de restaurer l’image à l’écran est le fait que l’on sérialise aussi le chemin de l’image sur le disque.

# **4.Conclusion**

En somme, le laboratoire 4 vise à mettre en place une application permettant d'afficher une image avec différentes perspectives possibles. Le client pourra appliquer sur la perspective les opérations suivantes : un zoom avant, un zoom arrière , un copier coller, sauvegarder son état mais aussi de pouvoir défaire une certaine commande.

Notre application à donc pour objectif de mettre en place une structure en utilisant le maximum de patrons de conception afin de rendre le code flexible et facilement modulable. L’utilisation des patrons de conception est la condition principale pour la mise en place de cette application tels que:

* Modèle Vue Contrôleur
* Commande
* Fabrique
* Mémento
* Observer
* Null Object
* Singleton
* Observer
* Immuable

Finalement après la mise en place de notre application, celle a permis de répondre aux besoins de l’application. Les patrons de conception qui ont été cités ci-dessus ont contribué à la concrétisation du projet. L’utilisation du patron de conception Modèle Vue Contrôleur a permis de rendre le code plus flexible. Ainsi, si on change la partie modèle ou contrôleur pour le traitement cela n’affecte nullement la vue. L’utilisation du patron de conception Commande nous a permis de mettre en place les différentes classes qui servent à la mise a jour des données dans le contrôleur dès que l’écouteur d’événements capte une action de utilisateur. L’utilisation du patron de Conception Mémento nous permet de sauvegarder l’état de notre perspective selon les transformations de celle-ci (classe : ImageSnap, Gardien, Originator) mais aussi de pouvoir les restaurer. L’implémentation du patron fabrique a permis de fournir au panneau d’image une instanciation de la classe Image sans passer par la classe d’origine. L’utilisation du patron de conception singleton nous offre ici une instance unique de la classe permettant d’établir une commande adéquate dès l'interception d’un événement depuis l’écouteur d’événements. Enfin, l’intégration du patron Observer permet dans notre cas dès modifications sur la perspective que celle-ci avertissement le panneau d’images par la commande notify() afin qu’il puisse se mettre à jour .

Même si notre application dispose de certaines petites faiblesses facilement améliorables, celle-ci permet de répondre à l’ensemble des besoins fournis dans l'énoncé du laboratoire.

Ce laboratoire nous a permis de prendre conscience de nombreux points tels que :

* L’utilisation des patrons de conceptions ainsi que leur intégration:
  + - Modèle Vue Contrôleur
    - Commande
    - Fabrique
    - Mémento
    - Observer
    - Null Object
    - Singleton
    - Observer
* Les diagrammes de classe UML ainsi que les diagrammes de séquences

Bien au delà, ce laboratoire nous a fait réaliser que lorsque on se lance dans un projet de programmation, il est primordial de s’attarder sur la partie conception qui est le pilier du développement futur. Si on néglige cette partie, il est évident que le projet sera entièrement désorganisé, voir même inachevé. Par conséquent, il faut impérativement analyser au préalable les besoins de l’application afin de formuler le diagramme de classe ainsi que leurs responsabilités afin de bâtir les diagrammes de séquences. Cette étape est donc la colonne vertébrale de l’application.

Ainsi, tous ces éléments nous ont été entièrement bénéfiques et nous permettront de les réutiliser dans des applications futures.

# **5.Références**

<http://fr.wikipedia.org/wiki/Singleton_(patron_de_conception)>

<http://en.wikipedia.org/wiki/Null_Object_pattern#Description>

<https://cours.etsmtl.ca/log121/private/fcardinal/>

<http://www.jmdoudoux.fr/java/dej/chap-serialisation.htm>