

Conception d'un système interactif en Java
MVC en MVC : Mitigeur Vitesse-Cap selon
l'architecture Modèle-Vue-Contrôleur (*2^{ème} partie*)

Mathieu Petit, Meriam Horchani

22 mai 2008

Résumé

Par rapport à d'autres langages, Java est livré avec des bibliothèques graphiques qui permettent l'implémentation de systèmes interactifs riches et performants. Deux d'entre elles sont plus communément utilisées : AWT (Abstract Window Toolkit) et son évolution multi-plateforme SWING. Cette bibliothèque structure tous ces composants selon le patron Modèle-Vue-Contrôleur, inventé dans les années 70 avec le langage SmallTalk. Vous verrez dans ce TD quels sont ses avantages pour la construction de systèmes interactifs.

Après les premiers tests, les utilisateurs sont encore frileux par rapport au prototype de pilotage mis en place. Pour surmonter cela, nous vous proposons d'imaginer de nouvelles solutions pour ce système interactif, qui dépasseront le modèle d'interface graphique orienté objet. A cette fin, vous appliquerez le concept du mitigeur pour simplifier l'arbre des tâches et permettre un dialogue plus efficace entre l'opérateur et le système de pilotage.

Table des matières

1	Ce que les utilisateurs ne peuvent savoir	2
1.1	Interfaces utilisateur orientées objet (IUOO)	3
1.2	Limites des IUOO	3
2	Principe du mitigeur	4
2.1	Mitigeur de dialogue	4
2.2	Mitigeur d'interaction	6

1 Ce que les utilisateurs ne peuvent savoir

Reprenons l'échange entre les marins et les concepteurs pour voir si le prototype proposé a fini par atteindre son objectif : être utilisé en lieu et place de l'ancien système de navigation.

[Après une première série de tests décevants sur un prototype qui n'affichait pas de retour numérique des valeurs de cap et de vitesse, les deux marins racontent leur expérience avec le nouveau système ... il semble que l'ancien système de commande ait toujours la cote.]

...

CONCEPTEUR 1: Vous n'utilisez pas plus le nouveau système que l'ancien ?

MARIN 1: Au début nous avons joué le jeu, c'est vrai que l'interface est moins austère que notre pupitre de commande

MARIN 2: Disons qu'on l'a utilisé pendant une semaine ...

MARIN 1: ... C'est déjà bien, non ?

CONCEPTEUR 2: Pour nous, ce sera bien quand vous aurez oublié l'ancien système !

CONCEPTEUR 1: Oui, le but c'est avant tout que vous soyez à l'aise avec ce que l'on vous propose. Arriveriez-vous à exprimer ce qui manque au prototype par rapport au système de commande ?

MARIN 1: On s'est posé la question, figurez-vous ... Objectivement, les deux systèmes fonctionnent et opèrent comme on leur demande.

MARIN 2: Je n'arrive pas à dire pourquoi, mais il est clair qu'à choisir, je préfère l'ancien pupitre de commande.

CONCEPTEUR 2: Je sèche ! [CONCEPTEUR 1 fouille son ancien grimoire d'IHM, à la recherche des notions de conception du dialogue d'un système interactif.]

CONCEPTEUR 1: Voilà, j'ai quelque chose. C'est vrai que nous avons beaucoup parlé de la tâche, et que notre analyse semble complète puisque le système est utilisable. Par contre, au niveau du dialogue avec le système, nous avons un peu fait l'impasse.

CONCEPTEUR 2: C'est vrai, je crois me rappeler que l'on est sensé analyser les problèmes au niveau du poste de commande pour cela.

MARIN 1: La je peux vous aider : le principal souci que l'on rencontre, c'est que les commandes de cap et de vitesse sont indiscernables [voir Fig. 1]

MARIN 2: C'est particulièrement vrai quand on doit réagir vite. On se trompe souvent ...

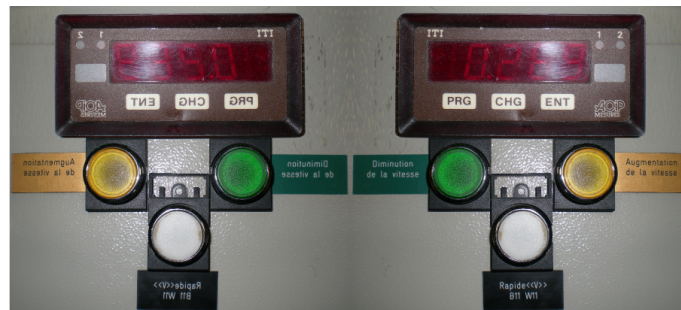




FIG. 1 – Poste de commande du bateau

1.1 Interfaces utilisateur orientées objet (IUOO)




Selon Jacob Nielsen, “*Object-oriented interfaces are sometimes described as turning the application inside-out as compared to function-oriented interfaces. The main focus of the interaction changes to become the users’ data and other information objects that are typically represented graphically on the screen as icons or in windows.*” (par rapport aux interfaces graphiques fonctionnelles, les interfaces orientées objet cherchent à rendre explicite l’application. Les opérations qui définiront les données et autres objets d’information des utilisateurs y sont représentées à l’écran sous forme d’icônes ou dans une fenêtre graphique)[1].

En manipulant une interface orientée objet, l’utilisateur agit explicitement sur des objets qui représentent (simulent) des concepts du domaine d’application. Cette notion est apparue au début des années 90 en réaction aux modèles d’interfaces utilisateur définissant une simple surcouche à un langage de commande (modèle du *Backend-Frontend*). Par exemple, un logiciel de traitement de texte comme Word, conçu autour du principe WYSIWYG, adopte une interface objet où la feuille virtuelle, représentée à l’écran, reprend et *enrichit* le modèle de la feuille de papier réelle. Les icônes représentent la fonction du système communément associée à l’objet réel :  pour imprimer,  pour agrandir. Dans la même veine, Windows95 généralise la notion de *bureau virtuel*, sur lequel on peut déposer des *dossiers* et des *documents*, qu’il est possible d’organiser par manipulation directe (mécanisme du *drag’n’drop*).

Question 1 *Ouvrez le complément du cours sur les interfaces objet et expliquez pourquoi les quelques exemples d’interfaces présentées sont orientées objet (ou pas).*

1.2 Limites des IUOO

Les interfaces orientées objets ont simplifié l’utilisation des systèmes interactifs, et ce malgré certaines limitations.

- Impossibilité de trouver un objet de référence : c’est le cas du copier/coller. Les icônes qui représentent ces actions ( et ) ont été choisies arbitrairement et ne sont utilisable que parce qu’un standard de fait s’est établi¹,
- tendance à se reposer sur les métaphores établies : le modèle numérique dépasse souvent les possibilités de l’objet physique représenté. Pour garder l’idée de l’objet sous-jacent, on enrichit artificiellement sa représentation, quitte à perdre l’analogie avec le réel qui faisait sa force. Par exemple, pour exprimer la notion purement numérique de “dé-zoomer”, on utilisera la représentation de l’outil zoom (); alors que dans la réalité, la loupe ne serait pas utilisée en pareille situation (bien au contraire!),
- biais de conception : des couches de présentation et de dialogue, qui ne font qu’imiter un objet réel, ne résolvent pas les éventuelles contraintes ergonomiques ou de design liées à la conception de l’objet représenté. Un objet réel complexe sera représenté comme un objet virtuel complexe. La conception de l’interface par imitation ou métaphore fait se reporter la réflexion sur la qualité du dialogue au niveau de l’objet physique,

¹Au passage, c’est ce qui vous interdit d’utiliser une autre représentation iconique pour ces actions.

- biais de manipulation : avec une métaphore de l’objet réel comme style d’interface, l’utilisateur serait en droit d’obtenir les mêmes qualités interactives que lors de la manipulation de l’objet réel. Ce n’est pas le cas. Les interfaces orientées objet sont une abstraction, à laquelle on accède par la médiation de la machine. Dans l’exemple de la platine virtuelle, l’utilisateur peut avoir l’impression de manipuler une platine physique, car les fonctionnalités représentées dans l’interface sont celles de la platine réelle, mais en fait, il est contraint par l’utilisation de la souris, la taille de l’écran, etc.

Question 2 *Remplacez le prototype MVC-stage1 par rapport aux biais et limitations des interfaces orientées objet. D’où provient la difficulté d’acceptation de ces prototypes par les marins ? Pourquoi peut-on dire que les concepteurs sont tombés dans le “biais de la conception” ?*

2 Principe du mitigeur

Pour limiter les biais liés aux modèles d’interfaces orientées objet, nous repartons de l’arbre des tâches existant en détaillant les actions d’interactions nécessaires pour effectuer l’ensemble de la tâche. Du point de vue du dialogue, cette séquence est réorganisée (éventuellement simplifiée) afin de se différencier de l’ordonnancement des opérations sur l’objet réel. En dernier lieu, le niveau interaction est modifié afin d’adapter la manipulation des concepts du domaine aux limites technologiques du système. Ces changements vont impliquer une redéfinition de la tâche où l’on passe d’une description opérationnelle liée au poste de commande à une description conceptuelle, non énoncée par les utilisateurs, mais que l’on espère plus proche des intentions et des besoins par rapport à la tâche abstraite considérée.

2.1 Mitigeur de dialogue

Chaque feuille de l’arbre des tâches peut-être détaillée en actions d’interaction pour l’utilisateur. Dans le prototype MVC-stage1, la sous tâche “Gérer le cap” consiste à sélectionner le cap, puis à l’incrémenter ou à le décrémenter. En généralisant, les prédicats définissant les primitives d’interaction sont alors :

$$\begin{aligned} a &= \{\text{Selectionner le cap}\} \\ b &= \{\text{Selectionner la vitesse}\} \\ c &= \{\text{Incrementer la valeur}\} \\ d &= \{\text{Decrementer la valeur}\} \end{aligned}$$

Si l’on exclut le retour d’information, la tâche $T = \{\text{Déplacer le bateau}\}$ se traduit en langage des prédicats par :

$$\begin{aligned} T &= T' \vee T'' \\ &= (a \wedge (c \vee d)) \vee (b \wedge (c \vee d)) \\ \text{avec } T' &= \{\text{Gerer le cap}\} \\ \text{et } T'' &= \{\text{Gerer la vitesse}\} \end{aligned}$$

Sans changer la tâche T , il est possible d'ordonnancer les primitives d'interaction de façon différente, c'est-à-dire de modifier le déroulement du dialogue avec l'utilisateur. Par exemple :

$$T = (a \vee b) \wedge (c \vee d)$$

Textuellement, cela signifie que la tâche “Déplacer le bateau” est aussi réalisable en sélectionnant soit le cap soit la vitesse, puis en incrémentant ou en décrémentant la valeur. Les sous-tâches “Gérer le cap” et “Gérer la vitesse” deviennent une conséquence de l'interaction et de fait n'appartiennent plus au niveau de la tâche (Fig. 2(a)).

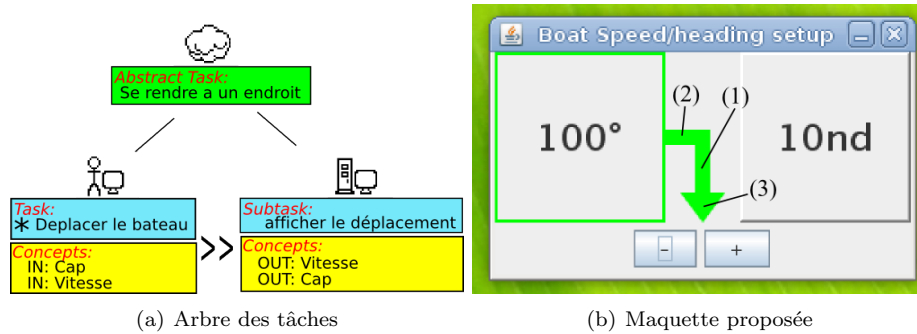


FIG. 2 – Système de navigation après l'application du mitigeur de dialogue

2.1.1 Conception de l'interface

Il n'y a plus de division nette entre cap et vitesse dans l'arbre des tâches. Pour l'architecture MVC, cela a deux conséquences :

- il n'est plus nécessaire d'avoir deux vues séparées pour le cap et la vitesse,
- un seul MVC est suffisant pour la gestion du déplacement.

Pour l'interface associée à la nouvelle organisation du dialogue, vous implémenterez la maquette proposée en Figure 2(b). L'archive **MVC-stage2.zip** sur Moodle contient le code du MVC agrégé et le début du code de l'interface graphique de la vue **MixedInputUIView**. Seules les parties contrôleur et vue ont été modifiées, le modèle est conservé par rapport à **MVC-stage1**.

Question 3 Charger l'archive *MVC-stage2.zip* dans *Éclipse* en tant que nouveau projet, éditez le source “*MixedInputUIView.java*” et suivez scrupuleusement ces opérations (les numéros des questions sont reportés dans le code source pour vous indiquer les endroits où il faut coder) :

1. dans la méthode `buildFrame()`, instanciez `controlPanel` avec le constructeur vide de `JPanel`,
2. ajoutez les boutons `moins` et `plus` au `controlPanel`,
3. instanciez `contentTop` avec le constructeur de `ModalJPanel(heading, speed)`,
4. ajoutez lui un `MouseListener` de lui-même,

5. ajoutez `contentTop`, puis `controlPanel` au `contentPane`.

Prenez quelques instant pour étudier la classe interne `ModalJLabel`. Celle-ci surcharge la classe Swing `JLabel` et définit sa décoration par rapport à un état (activé ou non) :

6. en commentaire, dans le code de la classe, décrivez le fonctionnement de la méthode `setSelected(boolean selected)`,
7. à quoi sert le `boolean selected` ?

La classe interne `ModalJPanel` construit un switcher. Elle instancie `l1` et `l2`, deux `ModalJLabel`, et `arrow`, de type `JPanel` qui contiendra le dessin de la flèche. Complétez d'abord le dessin du panel `arrow` en surchargeant la méthode `paintComponent()` :

8. Inspirez vous de l'exemple donné en annexe pour dessiner un rectangle vert vertical au milieu du `JPanel arrow` (Fig. 2(b)(1)); récupérez la hauteur et la largeur avec `this.getHeight()` et `this.getWidth()`,
9. dessinez un rectangle dont la position horizontale va dépendre de la valeur retournée par `l1.isSelected()` (Fig. 2(b)(2)),
10. optionnel et plus complexe : dessinez la pointe de la flèche (Fig. 2(b)(3)). Vous trouverez la méthode à la page <http://java.sun.com/docs/books/tutorial/2d/geometry/arbitrary.html>

Puis, dans le corps de la classe interne `ModalJPanel` :

11. créez deux méthodes `setLeftLabelText(String text)` et `setRightLabelText(String text)` qui initialisent les valeurs des champs texte des label `l1` et `l2` à `text`,
12. complétez la méthode `switchButton()` qui inverse l'état des `ModalJLabel l1` et `l2`, et qui repeint le `JPanel arrow`.

Il reste à connecter le code de réaction aux évènements sur les boutons de contrôle et sur le switcher, puis à réagir aux demandes de mise à jour du modèle :

13. complétez le code de réaction au clic sur le bouton “+” dans la classe interne `PlusActionListener` en vous inspirant de la classe interne `MinusActionListener`,
14. complétez le code de réaction au clic dans la classe interne `ModalJPanel`
15. modifiez les valeurs affichées dans les `ModalJLabel` lorsque la vue est mise à jour par le modèle (méthodes `headingChanged()` et `speedChanged()`)

Exécutez le code et vérifiez le comportement de l'interface.

2.2 Mitigeur d'interaction

Après un nouveau mois d'essais, le prototype commence à faire ses preuves. Les marins apprécient l'ordonnancement du dialogue, qui les “incite” à réfléchir au concept manipulé avant d'appuyer sur les touches de réglage. Visuellement, l'interface est claire et on identifie facilement la sélection en cours. Selon Marin 2, il s'agit “d'un vrai plus” par rapport au pupitre de commande.

Maintenant que le nouveau système trouve sa place sur la passerelle, les concepteurs ont tout le loisir d'observer les marins à l'oeuvre. Rapidement, ils repèrent deux comportements qui leur indiquent que le système proposé est encore améliorable :

- après un réglage de vitesse ou de cap sur le prototype ou sur l'ancien pupitre de commande, le marin va reporter ces valeurs sur la carte papier pour ajuster l'itinéraire du bateau jusqu'à l'endroit de destination,
- pour certaines manoeuvres, le marin va jouer simultanément sur le cap et la vitesse. Sur le pupitre de commande, il contrôle le cap de la main gauche, et la vitesse de la main droite. Cette opération n'est pas réalisable sur le système numérique, auquel on accède par la médiation de la souris (un seul dispositif).

2.2.1 Vue de contrôle du cap et de la vitesse

Pour économiser aux marins des aller-retours entre la carte papier et l'écran, on choisi de représenter la carte, la position et le cap pris par le bateau dans nouvelle vue² associée à la tâche système. La maquette à implémenter est donnée en figure 3. Le bateau est représenté par le centre de la rosace. Le cap est reporté

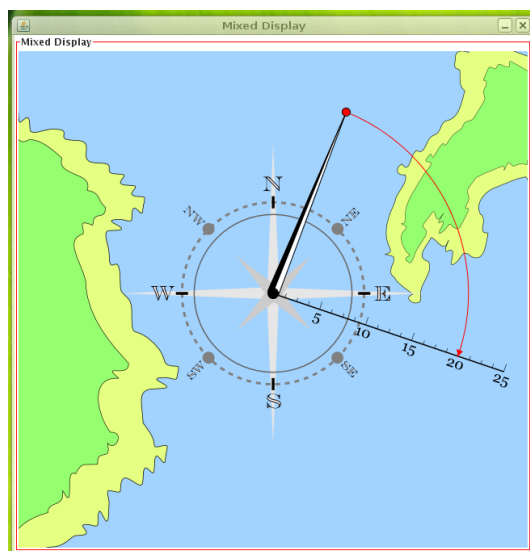


FIG. 3 – Mitigeur de présentation

par la direction de la flèche, la vitesse est donnée par la taille de la flèche et reportée par un arc sur une échelle de valeurs. Pour cet exemple, seule la flèche et l'arc rouge sont des éléments dynamiques. La carte, l'échelle de vitesse et la rosace sont intégrées dans une image de fond.

Dans l'archive **MVC-stage3.zip**, le contrôleur de la nouvelle vue et les méthodes de réaction aux événements de mise à jour ont déjà été codés. Pour cette partie d'implémentation, on se concentrera sur le contenu de la méthode `paintComponent()` de la classe interne `MixedDisplayPane`. Il vous est demandé de prendre un minimum d'autonomie, nous ne répondrons aux problèmes de dessins que si vous avez fait auparavant l'effort de fouiller le tutorial Java2D à la recherche d'une solution³.

²encore une interface orientée objet ...

³Tutorial Java2D : <http://java.sun.com/docs/books/tutorial/2d/index.html>

Question 4 Charger l'archive *MVC-stage2.zip* dans Éclipse en tant que nouveau projet. Éditez le source *"MixedInputUIView.java"* et suivez avec encore plus d'attention ces opérations (les numéros des questions sont reportées dans le code source pour vous indiquer les endroits où il faut coder). Pour commencer, on complète le constructeur de la classe interne *MixedDisplayPane* :

1. chargez l'image de fond avec `background=ImageIO.read(load(imageFile))`,
2. chargez le dessin du curseur (petit triangle rouge) avec `cursor=ImageIO.read(load("images/cursor.png"))`,
3. utilisez la méthode `setPreferredSize(Dimension d)` pour agrandir le *MixedDisplayPane* aux dimensions de l'image `background`.

La méthode `paintComponent` nous envoie en paramètre une "toile" (*Graphics2d g2d*) sur laquelle il est possible de dessiner n'importe quelle forme, image ou texte. Nous allons peindre l'image de fond dans l'objet *graphics2D* :

4. utilisez la méthode `drawImage()` de *g2d* pour dessiner l'image `background`,
5. exécutez le projet et vérifiez que l'image se charge correctement.

Le cercle rouge au bout de la flèche est positionnable assez simplement. Les coordonnées de son centre sont :

$$x = \text{speed} * \text{zoom} * \cos\left(\frac{(\text{heading} - 90) * \pi}{180}\right) + \frac{\text{largeur}}{2}$$

$$y = \text{speed} * \text{zoom} * \sin\left(\frac{(\text{heading} - 90) * \pi}{180}\right) + \frac{\text{hauteur}}{2}$$

6. enregistrez les coordonnées du centre dans deux variables *x* et *y* de type *Double*,
7. dessinez un disque rouge de rayon 5 centré en (*x*,*y*) (`g2d.fillOval()`),
8. dessinez un cercle noir de rayon 5 centré en (*x*,*y*) (`g2d.drawOval()`).

Pour définir l'arc, nous utiliserons la méthode `setArcByCenter` de la classe *Arc2D.Double*. On lui passe en paramètres le centre de l'arc, son rayon, l'angle du début de l'arc, l'amplitude angulaire, et le type d'arc *Arc2D.OPEN*. En démarant à -19° ⁴, l'amplitude angulaire α pour rejoindre le cercle rouge est donnée par :

$$\alpha = -((\text{heading} + 270 - 19) \bmod 360)$$

9. Instanciez un objet *arc* de type *Arc2D.Double*,
10. sur *arc*, appelez la méthode `setArcByCenter()` avec les bons paramètres,
11. dessinez cet arc sur la toile avec la méthode `g2d.draw(arc)`,
12. par un appel à `g2d.drawImage()`, dessinez l'image du curseur triangulaire à la position :

$$x = \text{speed} * \text{zoom} * \cos\left(\frac{19 * \pi}{180}\right) + \frac{\text{largeur}}{2} - 1$$

$$y = \text{speed} * \text{zoom} * \sin\left(\frac{19 * \pi}{180}\right) + \frac{\text{hauteur}}{2} - 9$$

⁴angle entre l'échelle des vitesses et l'axe des abscisses

Il reste encore à dessiner l'aiguille du cap, ce qui est un peu plus délicat. Il s'agit d'une superposition de 3 triangles : un triangle blanc en fond, une bordure de triangle noire de la même dimension et un demi triangle noir par dessus. Les coordonnées du triangle de fond blanc sont :

$$\left\{ \begin{array}{l} (x, y), \\ \left(7 * \cos \left(\frac{(heading-180)*\pi}{180} \right) + \frac{largeur}{2}, 7 * \sin \left(\frac{(heading-180)*\pi}{180} \right) + \frac{hauteur}{2} \right), \\ \left(7 * \cos \left(\frac{(heading)*\pi}{180} \right) + \frac{largeur}{2}, 7 * \sin \left(\frac{(heading)*\pi}{180} \right) + \frac{hauteur}{2} \right) \end{array} \right.$$

13. Créez la forme `GeneralPath triangle1` à partir des points donnés,
14. dessinez un triangle plein en blanc (`g2d.fill(triangle1)`),
15. dessinez le contour du même triangle en noir (`g2d.draw(triangle1)`),
16. créez le demi triangle noir dans `GeneralPath triangle2`. il est identique au `triangle1` sauf pour l'un des sommets qui est positionné sur l'origine,
17. dessinez `triangle2` en noir.

2.2.2 Principe du mitigeur d'interaction

Avec le modèle de carte numérique, le retour d'information est *explicite*. Il n'est plus nécessaire de contrôler sur la carte papier l'orientation prise par la navire. Reste que l'ancien poste de commande est toujours utilisé pour les manoeuvres synchronisées. Pour rendre l'utilisation du prototype aussi confortable, il faut remplacer le mitigeur de dialogue par un mitigeur d'interaction : une seule action d'interaction valide plusieurs étapes du dialogue et/ou de la tâche.

Question 5 *Imaginez un mitigeur qui permette de réaliser l'ensemble des tâches "Déplacer le bateau" et "Afficher le déplacement" en une seule étape d'interaction avec le système. D'ailleurs, est-ce possible ?*

L'analogie avec un robinet-mitigeur apparaît clairement : au prix d'un design un peu plus compliqué pour les concepteurs, l'utilisateur peut régler le débit et la température (deux concepts du domaine) par une seule action d'interaction sur le levier du mitigeur. Les modèles classiques de robinets permettent de commander efficacement le débit d'eau chaude et le débit d'eau froide, mais ils ne proposent qu'une réponse indirecte aux intentions des utilisateurs.

Question 6 *Détaillez l'arbre des tâches associé à "régler l'eau de la douche" pour la version du robinet à deux entrées. Faites de même pour le robinet mitigeur. Pouvez vous décrire la tâche "se rendre à un endroit" après l'application du mitigeur d'interaction sur le système de navigation ?*

2.2.3 Mise en oeuvre

Pour ce dernier prototype, nous partirons de la vue de contrôle mixte créé précédemment, et nous ajouterons la possibilité d'agir directement sur la pointe de la flèche pour positionner le curseur vitesse-cap directement au niveau de la carte. Par rapport à la classe `MixedDisplayUIView` de l'archive MVC-stage3, classe `MixedUIView` fournie dans l'archive MVC-stage4 implémente des `Listeners` sur des événements souris qui permettent de récupérer les actions de l'utilisateur sur la vue

Question 7 Chargez le projet, et ouvrez la classe *MixedUIView.java*.

1. dans la méthode *mousePressed()* de la classe interne *MixedDisplayPane*, si le curseur est dans un rayon de 5 autour du point (x, y) , passer le booléen *cursorSelected* à *true*,
2. dans la méthode *mouseReleased()* de la classe interne *MixedDisplayPane*, passer le booléen *cursorSelected* à *false* s'il est à *true*

Dans la methode *mouseDragged*, on mettra a jour les valeur *heading* et *speed* en fonction de la position du curseur à l'ecran avant de les envoyer au controleur pour qu'il mette à jour la vue. Autrement dit, vous n'avez pas à vous soucier de la façon dont la fenetre est redessinée.

La vitesse est donnée par :

$$speed = \frac{\sqrt{(\frac{largeur}{2} - arg0.x)^2 + (\frac{hauteur}{2} - arg0.y)^2}}{zoom}$$

Le cap est donné par :

$$heading = \begin{cases} \left(\pi - \sin^{-1} \left(\frac{arg0.y - hauteur/2}{speed * zoom} \right) \right) * \frac{180}{\pi} + 90 & \text{si } \frac{largeur}{2} > arg0.x \\ \left(\sin^{-1} \left(\frac{arg0.y - hauteur/2}{speed * zoom} \right) \right) * \frac{180}{\pi} + 90 & \text{sinon} \end{cases}$$

3. initialisez *heading* et *speed* aux nouvelles valeurs de cap et vitesse dans la methode *mouseDragged*
4. appelez la mise à jour par le contrôleur avec *getController().notifyHeadingSpeedChanged()*
5. executez le code et vérifiez que le fonctionnement du mitigeur cap/vitesse.

Références

- [1] J. Nielsen. Usability Engineering. Morgan Kaufmann, San Fransisco, CA, USA, first edition, 1993.

ANNEXE : Dessin d'un rectangle dans un JPanel

```
arrow=new JPanel(){  
    public void paintComponent(Graphics g){  
        Graphics2D g2d = (Graphics2D)g;  
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,  
            RenderingHints.VALUE_ANTIALIAS_ON);  
        //suite du code de dessin  
        //Rectangle vert de 10x20 au point x=5 y=10  
        g2d.setColor(Color.GREEN);  
        g2d.fillRect(5,10,10,20);  
    }  
}
```