TP6 et TP7 : IHM du jeu de Morpion en *Swing*

Le jeu de Morpion est un jeu à deux qui se joue sur un damier de taille 3 cases sur 3. Les cases du damier sont initialement vides. Chaque joueur joue en alternance en inscrivant un symbole dans une case vide (l'un des joueurs est représenté par un anneau ou un rond, l'autre joueur par une croix). Le jeu s'arrête dès que l'un des joueurs aligne trois symboles identiques (horizontalement, verticalement ou diagonalement) ou lorsque toutes les cases sont occupées.

Exercice 1 : La bibliothèque *Swing*

Swing est une bibliothèque graphique conçue en et pour Java, avec pour objectif de permettre une expérience utilisateur identique quel que soit le système sous-jacent (Windows/Mac/Linux…). Elle contient principalement des objets graphiques interactifs (widgets), de manière similaire à ceux que vous avez manipulés avec *Qt* en 1ère année.

- Créez un nouveau projet Java sans classe principale (décochez l'option "Create Application Class") et importez y le code de l'archive *SwingDemo.zip*. Exécutez l'application, puis observez attentivement son code source et les commentaires qui y sont rédigés.
- Consultez l'URL « *A Visual Guide to Layout Managers* » sur e-campus pour avoir un aperçu des widgets disponibles.

Exercice 2 : Le jeu de Morpion

2.1 Préparation

Créez un nouveau projet "TP6" sans classe principale puis copiez le code et les ressources fournis avec le sujet en suivant les indications ci-dessous :

- coller le contenu du dossier "src" dans le dossier source du projet,
- coller le dossier "resources" à la racine du projet (il contient des images à afficher).

Le code fourni contient une implémentation fonctionnelle en mode textuel. Dans ce TP vous allez développer une deuxième version, en *Swing*. Et vous allez le faire facilement car le code est architecturé en suivant un certain nombre de **patrons de conception** (en anglais ***design patterns***), c'est à dire des méthodes/bonnes pratiques qui permettent de traiter des problèmes logiciels généraux et facilitent la maintenabilité du code.

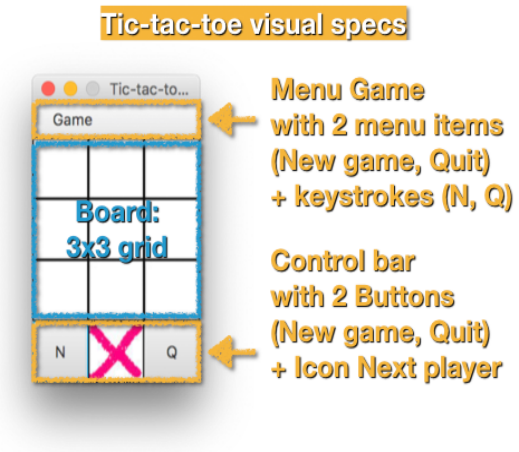
Après avoir vérifié que l'application fonctionne bien sur votre ordinateur lisez en le code et sa documentation (vous pouvez même générer la javadoc en faisant un clic droit sur le nom du projet et en choisissant l'item "*Generate Javadoc*"). Examinez également les diagrammes de classe et de séquence en annexe pour mieux comprendre l'architecture et le fonctionnement du code, puis répondez aux questions suivantes :

- Quels sont les patrons de conception utilisés ?
- Pourquoi sont ils utilisés ?
- Lequel d'entre eux permet de basculer facilement entre chaque implémentation ?
- Lequel d'entre eux facilite un changement éventuel de technologie pour l'IHM (l'introduction de la version *Swing* en remplacement de la version texte) sans toucher au noyau fonctionnel ?

Nous allons maintenant développer la version *Swing* en plusieurs étapes : créer les widgets et le layout, ajouter une interactivité basique, lier l'ensemble au noyau fonctionnel, et enfin augmenter l'utilisabilité en ajoutant des menus, raccourcis clavier…

2.2 Layout

Conformément à la spécification visuelle ci-dessous la fenêtre de l'application (classe `JFrame`) doit être capable d'aligner ses enfants verticalement, le conteneur du plateau de jeu doit les arranger dans une grille, et la barre de contrôle doit les aligner horizontalement. De façon similaire à ce qui se fait en *Qt*, cet alignement passe par l'affectation d'un layout au conteneur. Identifiez les meilleurs candidats pour composer les différents conteneurs du jeu dans le code fourni pour l'exercice 1 et dans la doc *Swing* sur les layouts (https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html).



En vous inspirant du code fourni pour l'exercice 1 créez maintenant votre classe `TicTacToeSwingView` et peuplez la des widgets nécessaires :

- les conteneurs que vous avez choisis,
- des instances de `JButton` pour les boutons,
- des instances d'`ImageIcon` pour charger les images (mais pas les afficher),
- des instances de `JLabel` pour les cases de la grille et la case "Next player" (et non des `JButton` pourquoi?),
- n'oubliez pas que comme en *Qt*, il ne suffit pas de créer les widgets pour qu'ils s'affichent. Il faut aussi penser à les ajouter à leur parent avec la méthode `add()`.

Testez votre application (n'oubliez pas de basculer vers la vue *Swing* dans la classe principale) et réglez les problèmes d'alignement. Pour ce faire vous devrez avoir des boutons de la même taille que la case "Next player" (recherchez les méthodes appropriées dans la javadoc de `JButton` et `JLabel`).

2.3 Interactivité

Les abonnements pour pouvoir réagir aux événements (provoqués ici par un clic sur un bouton ou sur un label) fonctionnent de façon similaire à *Qt* (connexion *signal/slot*), mais avec une syntaxe plus compliquée :

- l'équivalent du *signal*, l'événement n'est plus un attribut du widget que l'on souhaite observer mais une classe (voir la doc https://docs.oracle.com/javase/8/docs/api/java/awt/event/package-summary.html) → pour une action sur un bouton (un clic) l'événement est du type `ActionEvent`,
- l'équivalent du *slot* n'est plus une simple fonction mais une classe implémentant une interface spécifique à l'événement → ici `ActionListener`, contenant une (ou plusieurs) méthode à redéfinir, ici `actionPerformed(…)`, dans laquelle vous placerez le code à exécuter en réaction à l'événement,
- l'équivalent de la *connexion* se fait en invoquant sur le widget une méthode également spécifique à l'événement → ici `addActionListener(…)`.

A la lumière des indications précédentes et en vous inspirant du code fourni pour l'exercice 1 commencez par les boutons. Les réactions aux actions sur les boutons seront dans un premier temps un message intelligible dans la console.

Pour les cases, les `JLabel` n'émettant pas d'`ActionEvent` il y a quelques différences :

- l'événement à observer est maintenant `MouseEvent`,
- le listener à utiliser est `MouseListener`. Lisez sa javadoc, expliquez pourquoi cette solution est laborieuse dans notre cas (elle ne l'est pas toujours), et proposez une autre stratégie,
- l'abonnement pour un `MouseListener` se fait en invoquant `addMouseListener(…)`.

Une fois la méthode d'abonnement choisie vous avez 2 stratégies pour réaliser l'abonnement sur les cases :

- soit réaliser un abonnement différent pour chaque case, auquel cas il vous suffit d'imprimer (dans un premier temps) les indices de la case dans la console,
- soit factoriser le code dans un `MouseListener/MouseAdapter` unique, auquel cas il faudra récupérer la source de l'événement pour identifier la case puis retrouver ses indices dans un tableau de `Jlabel` à 2 dimensions, puis imprimer ces indices dans la console. Cette solution étant de toute évidence plus compliquée, pour quelle raison et dans quel cas serait elle plus judicieuse ?

Faites votre choix et implémentez le.

Avec ces abonnements vous avez presque terminé le développement de la partie *contrôleur*. Pour rendre le jeu fonctionnel il reste juste à lier ce *contrôleur* au *noyau fonctionnel* existant.

2.4 Lien au noyau fonctionnel

Remplacez maintenant les `print` dans la console par les bons appels de méthodes dans le *modèle*. Vous pouvez vous aider en examinant le diagramme de séquence et la version texte. Conformément au diagramme de séquence le *modèle*, une fois mis à jour, calculera les résultats du coup qui vient d'être joué et appellera ensuite les bonnes méthodes de la vue pour la mettre à jour à son tour…

...Et pour que votre vue puisse être mise à jour par le modèle, la toute dernière étape consiste à faire comme dans la version texte : votre classe `TicTacToeSwingView` doit implémenter l'interface `IBoardGameView`. Vous devrez alors implémenter les méthodes qui sont spécifiées dans l'interface.

Indications :

- pour la méthode `displayGame(…)` il ne faut pas recréer la grille mais juste mettre à jour les images affichées par les `JLabel`,
- pour la méthode `displayLastMove(…)` il faut juste mettre à jour l'image spécifiée puisque d'un coup à l'autre, il n'y a qu'une case qui change,
- pour les autres méthodes utilisez la méthode de classe `showMessageDialog()` de la classe `JOptionPane` qui est le composant *Swing* le plus adapté.

Notre *vue Swing* est maintenant fonctionnelle. La dernière question est consacrée à l'implémentation de quelques services standards pour améliorer l'utilisabilité de toute application interactive.

2.5 Amélioration de l'utilisabilité

Même si dans notre jeu toutes les actions sont déjà directement accessibles par les cases et les boutons nous allons mettre en place des **menus** :

- La barre de menu se met en place en créant une instance de la classe `JMenuBar`, et en l'associant à votre `JFrame` grâce à sa méthode `setJMenuBar(…)`,
- un menu se met en place en créant une instance de la classe `JMenu`, et en l'associant à votre `JMenuBar` grâce à sa méthode `add(…)`,
- un item de menu se met en place en créant une instance de la classe `JMenuItem`, et en l'associant à votre `JMenu` grâce à sa méthode `add(…)`,
- un item de menu émet un événement `ActionEvent` lorsqu'il est sélectionné.

Comme `JButton` et le `JMenuItem` émettent le même événement et qu'on veut la même réaction, factorisez cette réaction dans un seul `ActionListener` par fonction (par exemple pour quitter) ou encore mieux un unique pour toutes les fonctions.

Enfin dans toutes les applications, afin d'augmenter l'efficacité des utilisateurs experts, en plus des menus, les plus importantes fonctions devraient toujours être accessibles par des **raccourcis clavier** (en anglais **keyboard shortcuts**) :

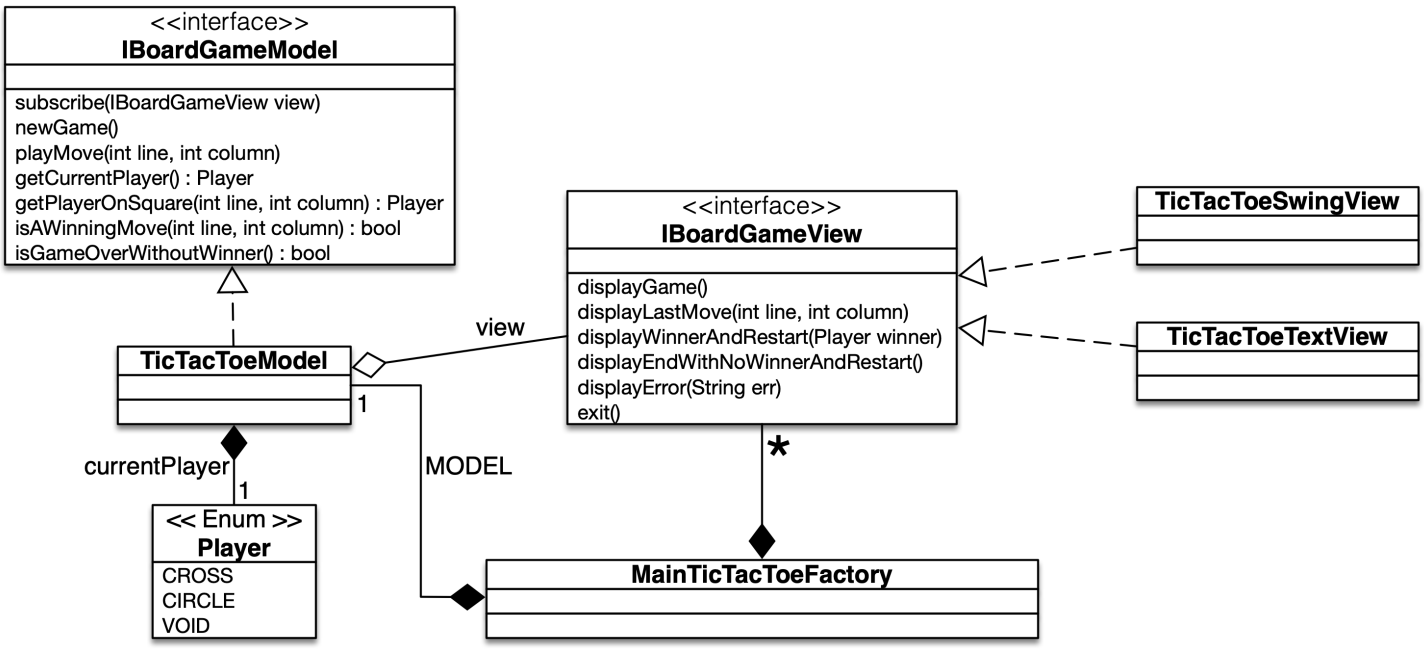
- créez une instance de la classe `KeyStroke` (regardez dans sa javadoc),
- associez la à un `JMenuItem` grâce à sa méthode `setAccelerator()`.

Exercice 3 (pour les kings of Java) : multi-vues et jeu en réseau

Si vous avez fini les TP précédents (tous ! Sinon vous savez ce qu'il vous reste à faire…) :

- Modifiez le modèle pour qu'il fonctionne simultanément avec plusieurs vues.
  Indications : remplacer l'attribut vue par une `ArrayList`, l'alimenter depuis la méthode souscrire() et faire les notifications aux vues avec un `forEach(Consumer…)` sur l'`ArrayList`
- Ajoutez la possibilité de sauvegarder/charger une partie en cours.
  Indications : il faut « sérialiser » l'état du modèle (cf. votre moteur de recherche préféré et doc *Java*)
- Ajoutez la possibilité de jouer en réseau (en utilisant le bus *Ivy*, donc uniquement en réseau local).

Annexe 1 : Diagramme de classes



Annexe 2 : Diagramme de séquence