V1.5

PROGRAMMATION ORIENTEE OBJET

Romuald GRIGNON

Coding Factory by ITESCIA
CCI Paris Ile-de-France
Cergy-Pontoise / Paris-Champerret

Année scolaire 2019-2020

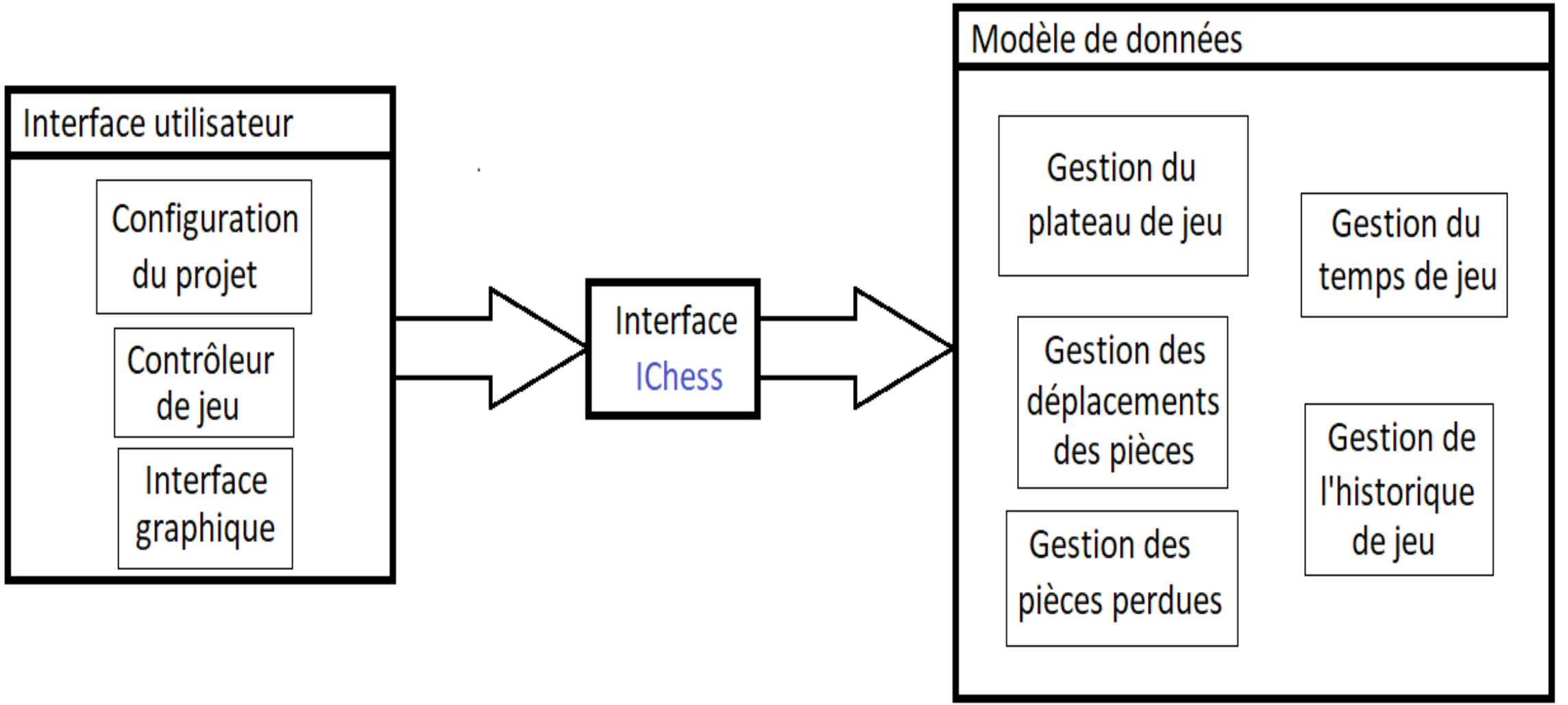
APPLICATION DE JEU D'ECHECS

Présentation du projet

Contexte

- Vous travaillez dans une société qui développe des applications informatiques.
- Un client a commandé une application 'desktop' de jeu d'échecs.
- L'équipe de développement est divisée en deux : l'une qui va prendre en charge l'interface utilisateur et le séquencement du jeu, et l'autre, le modèle de données.
- Pour pouvoir développer de manière la plus autonome possible, une interface a été définie entre les deux équipes.
- Vous êtes affecté à l'équipe qui va coder le modèle de données. L'équipe de l'interface utilisateur, quant à elle, a déjà terminé son développement.
- Afin d'assurer la maintenabilité du code et son évolution, des patrons de conception spécifiques vont être utilisés dans le modèle de données.
- L'application est codée en langage Java.
- L'environnement de développement logiciel utilisé par la société est IntelliJ.
- Vous travaillerez en mode Agile-Scrum, les stories fournies par le P.O. ayant déjà été classées en accord avec les priorités du client.

Schéma simplifié de l'application



Backlog stories

Informations générales

- Pour toutes les stories, les critères d'acceptation suivants s'appliquent (en plus des critères explicitement définis pour chaque story) :
  - La réalisation d'une fonctionnalité ne doit pas apporter de régression aux fonctionnalités précédentes.
  - Le code doit compiler et s'exécuter.
  - Respecter les bonnes pratiques Java pour le nommage des éléments du code.
  - Les noms des variables, méthodes, classes ainsi que les commentaires, doivent être en anglais.
  - Chaque élément de votre code doit être documenté au format JavaDoc.
  - Le code doit être archivé sous gestion de configuration (Mercurial, Git, SVN, CVS, …)

- Les règles du jeu d'échecs peuvent être récupérées sur la page Wikipedia :
  - https://en.wikipedia.org/wiki/Chess

Story #1 : Intégration initiale

- En tant qu'étudiant de la Coding Factory,
- Je veux faire compiler le projet de jeu d'échecs
- Afin d'avoir une base de travail initiale pour créer le jeu complet.
- Critères d'acceptation :
  - Récupérer le projet initial [https://bitbucket.org/Romuald78/chesspublic]
  - Créer dans le package game une classe ChessModel qui implémente une version minimale (méthodes 'vides') de l'interface IChess.
  - Modifier la classe ChessModel pour qu'elle intègre le patron de conception 'singleton'.
  - Le plateau de jeu doit apparaître.

Story #2 : Création des pièces

- En tant qu'étudiant de la Coding Factory,
- Je veux commencer à créer les données de la partie du jeu d'échecs
- Afin de pouvoir visualiser les pièces sur le plateau.
- Critères d'acceptation :
  - Créer une classe Piece pour les pièces du jeu, qui contiendra des attributs privés de type ChessColor et ChessType ainsi que les méthodes publiques pour y accéder.
  - Ajouter un attribut à votre classe ChessModel qui contiendra les données du plateau de jeu en mémoire.
  - Remplir le modèle de données à l'aide des classes créées précédemment (placer les pièces à leurs positions initiales).
  - Implémenter les méthodes d'interface getPieceColor et getPieceType pour que les données de votre modèle puissent être affichés à l'écran.
  - Les pièces du jeu d'échecs doivent apparaître sur le plateau à leurs positions initiales.

Story #3 : Pièces restantes

- En tant qu'étudiant de la Coding Factory,
- Je veux pouvoir récupérer le nombre de pièces restantes par couleur
- Afin de l'afficher sur l'écran.
- Critères d'acceptation :
  - Nouvelle implémentation de la méthode getNbRemainingPieces qui doit compter le nombre de pièces présentes sur le plateau de jeu.
  - Le nombres de pièces restantes doit s'afficher et doit être cohérent avec les images des pièces affichées sur le plateau.

Story #4 : Déplacements possibles

- En tant qu'étudiant de la Coding Factory,
- Je veux pouvoir récupérer les déplacements possibles de chaque pièce
- Afin de visualiser quelles sont les pièces que je peux jouer ou non.
- Critères d'acceptation :
  - Intégrer le patron de conception 'Strategie' dans la classe Piece. Pour cela, créer une interface IMove et les classes de chaque type de pièce qui doivent l'implémenter.
  - Implémenter la méthode d'interface getPieceMoves qui doit s'appuyer sur le nouveau code de l'interface IMove.
  - Chaque pièce qui peut se déplacer d'au moins une case sera entourée d'un halo.
  - En cliquant sur une pièce entourée d'un halo, on la sélectionne, faisant apparaître les cases de déplacements possibles pour cette pièce.
  - Les cases de déplacements possibles doivent être en accord avec les règles du jeu d'échecs.
  - Le « roque » du roi ne fera pas partie des déplacements possibles dans cette story.
  - La «prise en passant » du pion ne fera pas partie des déplacements possibles dans cette story.

Story #5 : Déplacement normaux

- En tant qu'étudiant de la Coding Factory,
- Je veux pouvoir déplacer les pièces sur le plateau de jeu
- Afin de pouvoir commencer à jouer réellement aux échecs avec cette application.
- Critères d'acceptation :
  - Implémenter la méthode d'interface movePiece pour déplacer les pièces dans le modèle de données.
  - Les pièces doivent pouvoir se déplacer sur l'échiquier et manger les pièces adverses.
  - Comme la notion d'échec au roi n'est pas codée, à ce stade on peut donc manger n'importe quelle pièce de l'adversaire, même le roi, et le jeu doit pouvoir continuer à s'exécuter malgré tout.

Story #6 : Déplacements spéciaux

- En tant qu'étudiant de la Coding Factory,
- Je veux pouvoir déplacer les pièces de manière spéciale
- Afin d'intégrer tous les déplacements des jeux d'échecs.
- Critères d'acceptation :
  - Modifier le code pour rajouter le roque du roi (attention à bien modifier l'implémentation de IMove pour le roi ET la méthode movePiece en accord avec ce déplacement particulier).
  - Modifier la méthode movePiece pour transformer un pion en dame quand il arrive au bout de l'échiquier dans le camp adverse.
  - Les pièces doivent pouvoir se déplacer sur l'échiquier et manger les pièces adverses.
  - Comme la notion d'échec au roi n'est pas codée, à ce stade on peut manger n'importe quelle pièce de l'adversaire, même le roi, et le jeu doit pouvoir continuer à s'exécuter malgré tout.
  - Rappel : la notion de «prise en passant » n'est toujours pas à traiter dans cette story.

Story #7 : Statut du roi

- En tant qu'étudiant de la Coding Factory,
- Je veux récupérer le statut du roi de chaque couleur
- Afin de pouvoir l'afficher sur l'interface.
- Critères d'acceptation :
  - Modifier la méthode d'interface getKingState afin de retourner l'état du roi de chaque couleur.
  - Cette méthode doit simplement vérifier si le roi est en échec par une pièce adverse ou non.
  - Le statut du roi est affiché sur l'écran et doit refléter la position des pièces sur le plateau de jeu.
  - A ce stade, on peut toujours se déplacer librement sans être contraint par l'échec au roi. C'est juste une information qui est affichée.

Story #8 : Limitation des déplacements

- En tant qu'étudiant de la Coding Factory,
- Je veux limiter les déplacements des pièces en fonction du statut du roi
- Afin de réellement appliquer les règles d'un jeu d'échec.
- Critères d'acceptation :
  - Il n'est pas autorisé de laisser le roi de sa couleur en échec après un déplacement.
  - Modifier le code du modèle de données afin de ne pas permettre des coups interdits. Notamment, la méthode d'interface getPieceMoves renverra une liste plus restreinte des positions possibles pour chaque pièce en fonction des positions des rois.
  - Si une couleur ne peut plus effectuer de mouvement sans laisser son roi en échec, la partie sera terminée.
  - On ne va pas tenir compte du fait que les cases par lesquelles passe le roi pendant le roque ne doivent pas être menacées par l'adversaire : seule la destination finale après mouvement sera gérée dans cette story.
  - A ce stade, toutes les règles des échecs sont implémentées. L'application est complète du point de vue des règles de déplacement (hormis le cas particulier du roque énoncé ci-dessus et la « prise en passant »).

Story #9 : Pièces perdues

- En tant qu'étudiant de la Coding Factory,
- Je veux pouvoir récupérer la liste des pièces qui ont été perdues par chaque camp

  Afin de les afficher à l'écran.
- Critères d'acceptation :
  - Implémenter la méthode d'interface getRemovedPieces afin de retourner la liste des pièces perdues depuis le début de la partie pour chaque camp.
  - C'est lors de l'appel à la méthode movePiece que l'on pourra récupérer l'information si une pièce a été perdue, et donc que l'on pourra la stocker.
  - Les pièces qui sont perdues au cours de la partie doivent s'afficher sur la gauche de l'écran.

Story #10 : Réinitialisation d'une partie

- En tant qu'étudiant de la Coding Factory,
- Je veux pouvoir réinitialiser le modèle de données
- Afin de recommencer une partie depuis le début.
- Critères d'acceptation :
  - Modifier la méthode d'interface reinit pour réinitialiser le modèle de données.
  - Après un appui sur la touche ENTER, ou clic gauche sur le bouton « STOP », le plateau de jeu a été complètement réinitialisé, prêt à démarrer une nouvelle partie.

Story #11 : Annuler un coup déjà joué

- En tant qu'étudiant de la Coding Factory,
- Je veux disposer d'une fonctionnalité qui permet d'annuler un coup
- Afin de pouvoir revenir dans un état précédent si j'ai déplacé une mauvaise pièce (ou si je souhaite tricher ^^).
- Critères d'acceptation :
  - Modifier la méthode d'interface undoLastMove pour être en mesure de revenir à un état précédent du jeu.
  - Implémenter au moins l'annulation du dernier coup.
  - Attention à la compatibilité avec les déplacements spéciaux comme le roque du roi ou la transformation d'un pion en dame.
  - L'appui sur la touche BACKSPACE, ou un clic gauche sur le bouton « PREVIOUS » effectuera l'annulation du dernier coup.
  - [Bonus] Si possible, implémenter l'annulation de tous les coups.

Story #12 : Mesure du temps de jeu

- En tant qu'étudiant de la Coding Factory,
- Je veux mesurer le temps de jeu de chaque joueur
- Afin de pouvoir afficher ce temps sur l'interface.
- Critères d'acceptation :
  - Modifier la méthode d'interface getPlayerDuration afin de récupérer le temps de jeu pour chacun des joueurs.
  - C'est l'appel à la méthode d'interface movePiece qui permet de synchroniser le modèle sur la mesure du temps de chaque joueur.
  - Cette fonctionnalité doit être compatible avec l'annulation des coups (c'est à dire qu'en annulant un coup, il faut revenir au temps de jeu du coup précédent).
  - La vérification du temps de jeu de chaque joueur se fait sur l'interface graphique.

Story #13 : [Bonus] Finalisation du roque

- En tant qu'étudiant de la Coding Factory,
- Je veux finaliser la gestion du roque du roi
- Afin de pouvoir (presque) respecter toutes les règles du jeu d'échec.
- Critères d'acceptation :
  - Modifier le code afin de ne pas accepter de roquer le roi si l'une des cases par lesquelles passe le roi est menacée par l'adversaire. Cela signifie aussi que le roque ne peut pas servir à échapper à une mise en échec.

Story #14 : [Bonus] « Prise en passant »

- En tant qu'étudiant de la Coding Factory,
- Je veux implémenter la règle « prise en passant »
- Afin de pouvoir (enfin) respecter toutes les règles du jeu d'échec.
- Critères d'acceptation :
  - Modifier le code afin de proposer un coup supplémentaire au pion de l'adversaire dans le cas de la « prise en passant ».
  - Attention à la fonctionnalité « retour en arrière ».
  - A ce stade, l'application est complète. Si l'ensemble des stories est accepté, cela signifie que le client est satisfait à 100% → GG ! EZ Noob ! Commend me !