Projet dirigé : jeu de Morpion

support des activités de réalisation

Jonathan ILIAS-PILLET,
Richard WOODWARD

Département Informatique et Systèmes

2016–2017

GROUPE ESEO
10, boulevard Jeanneteau – CS 90717 – 49107 ANGERS Cedex 2 – Tél : 02 41 86 67 67

1 Introduction

Ce document va vous guider dans la réalisation d'un jeu de Morpion. L'objectif est ici de vous donner quelques « recettes » de fabrication d'un tel jeu, avant de faire le votre. Ainsi, certaines parties (les plus triviales) vous sont données afin que vous puissiez vous concentrer sur les parties les plus intéressantes (pédagogiquement parlant).

1.1 Conception : architecture

Nous avons déjà structuré ce jeu en plusieurs modules ainsi que décrit dans la figure 1.

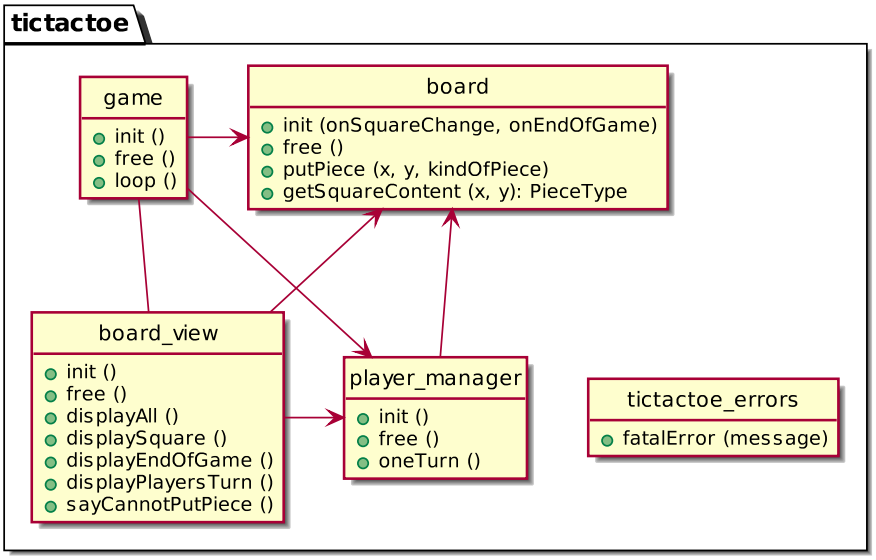

FIGURE 1 – Architecture du jeu de Morpion

Les modules *game* et *board* constituent la base logique du jeu (essentiellement les règles du jeu), réutilisable quelle que soit l'interface développée. Dans un premier temps, c'est sur cet aspect du jeu que nous allons concentrer nos efforts. Comme il est d'usage en développement logiciel, nous développons d'abord un moteur solide avant de nous faire plaisir à peaufiner la carrosserie.

Le module *board_view* comprend tout ce qui est nécessaire à l'affichage à destination de l'utilisateur (la carrosserie).

Le module *player_manager* est l'interprète des actions de l'utilisateur, traduisant ainsi les modalités techniques (clic souris, saisie clavier) en action logique sur le jeu (le volant et les pédales pour revenir sur la métaphore automobile).

1.2 Conception : interactions entre les modules

Le diagramme de séquence présenté en figure 2 présente une situation où un joueur joue un coup et que la partie se termine. Peu nous importe ici les paramètres effectivement échangés ou le résultat de la partie, il s'agit de mieux comprendre le rôle de chaque module vis-à-vis des autres.

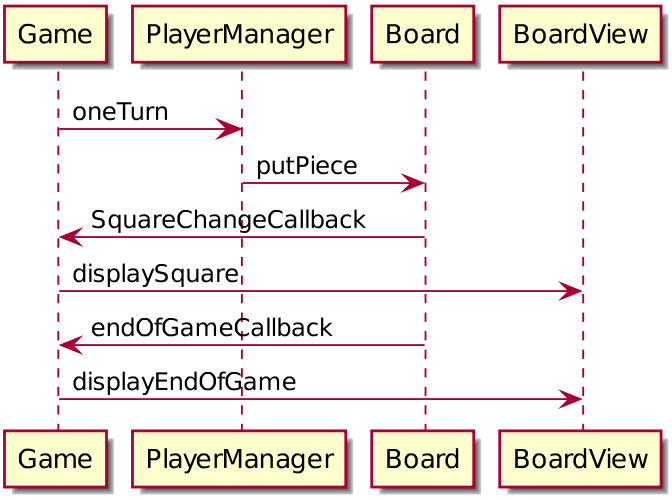

FIGURE 2 – Séquence d'un coup provoquant la fin de partie

Dans cet exemple, la demande d'un coup émane du module *Game* mais c'est dans la fonction `oneTurn` de *PlayerManager* que l'action de l'utilisateur est effectivement attendue et qu'elle est engagée après du module *Board*.

Ce dernier appelle successivement les fonctions `SquareChangeCallback` et `EndOfGameCallback` dont les adresses ont été renseignées à l'initialisation du module *Board*. Elle correspondent à des fonctions du module *Game* qui a la charge de coordonner l'ensemble du jeu. En l'occurrence, ces fonctions appellent les fonctions respectives du module d'affichage *BoardView*.

1.3 Règles de réalisation

1.3.1 Nommage des modules et fonctions

Pour traduire la conception résumée ci-dessus, nous nommons :

- les fichiers comme les modules (*board_view.h* et *board_view.c* pour le module *board_view*) ;
- les fonctions en les préfixant avec le nom du module (`PlayerManager_oneTurn` pour la fonction `oneTurn` du module *player_manager*).

Notez l'utilisation de la notation dite « CamelCase » dans les noms de fonction (avec le caractère « underscore » pour séparer le nom du module du nom de la fonction). Pour les fichiers, nous avons retenu la notation exclusivement en minuscule avec « underscores » pour séparer les mots [1].

1. En effet, sous UNIX, l'usage veut que les noms de fichiers soient, sauf rares exceptions, écrits uniquement en

1.3.2 Documentation et Doxygen

La documentation externe des modules, types, constantes et fonctions publics est disponible dans les fichiers d'en-tête respectifs (« *.h* »).

Quoi de plus naturel, pour coder une fonction, que de connaître les modalités précises du service à rendre ? Il est donc indispensable que vous preniez connaissance de la documentation des fonctions que vous allez réaliser (les indications du présent document ne sont qu'un complément destiné à vous orienter dans les actions de réalisation).

Si vous souhaitez une lecture plus confortable de cette documentation, vous pouvez extraire la documentation Doxygen en vous basant sur le fichier de configuration (*Doxyfile*) fourni avec le code source sur le campus.

1.3.3 Gestion de configuration et compilation conditionnelle

Dans la suite de ce document, vous aurez à remplacer une implémentation d'un module par une autre (typiquement pour *board_view*, la carrosserie). Dans ce cas, nous optons pour que le choix de la configuration (c'est à dire la sélection de l'implémentation pour chaque module) se fasse au moment de la construction. En langage C, cela passe par l'utilisation des directives de précompilation telles que « `#if defined MACRO` ». Le code source fourni sur le campus est déjà préparé à cet effet ; il ne reste plus qu'à paramétrer correctement la construction.

Dans Eclipse, vous pouvez définir une macro à la construction dans les paramètres du projet (par exemple « `CONFIG_TEXTUI` ») : « C/C++ Build → Settings → Tool Settings → GCC C Compiler → Symbols ».

2 Étape 1

2.1 Objectifs généraux

Dans un premier temps, vous allez vous concentrer sur le développement de la logique du jeu, donc principalement les modules *board* et *game*.

En vue de valider la logique du jeu, il faudra « bouchonner »les interfaces (modules *board_view* et *player_manager*).

2.2 Description du travail à réaliser

L'ordre de réalisation vous est donné ci-après, avec quelques informations utiles le cas échéant.

Vous trouverez le code source de base de l'étape 1 sur le campus.

minuscule

2.2 Description du travail à réaliser

2.2.1 Fonction `isGameFinised`

Écrivez le corps de cette fonction en vous aidant de la documentation Doxygen de la fonction (dans *board.c*) et du schéma de fonctionnement présenté en figure 3.

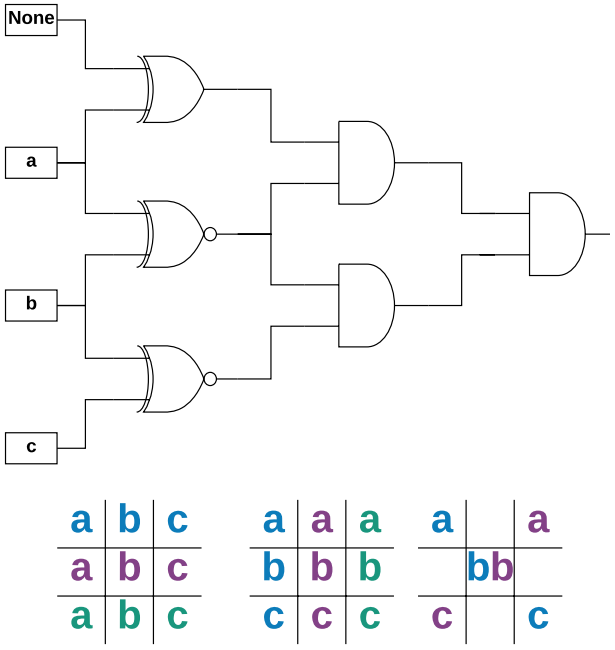

FIGURE 3 – Fonctionnement de `isGameFinished`

Le code source d'une suite de tests unitaires de cette fonction est fourni sur le campus. Vous pouvez l'utiliser et également l'étendre pour procéder à la validation de cette fonction.

2.2.2 Fonction `Board_putPiece`

Cette fonction doit s'assurer que la case demandée est bien vide (voir documentation Doxygen).

Lorsque le pion (croix ou cercle) spécifié est placé, il faut vérifier si la partie est terminée en utilisant `isGameFinished`.

Il faudra également appeler les fonctions `SquareChangeCallback` et `EndOfGameCallback` (adresses fournies lors de l'appel à `Board_init`) au moment adéquat.

2.2.3 Fonctions `Board_getSquareContent`, `Board_init` et `Board_free`

La documentation de ces fonctions devrait être suffisante pour vous permettre de compléter le corps de ces fonctions.

2.2.4 Module *board_view_text*

Complétez l'ensemble des fonctions de ce module en vous appuyant sur des fonctions type `printf` pour effectuer les affichages demandés (voir documentation Doxygen disponible dans *board_view.h*.

Remarque pour la construction

Notez qu'il faut ajouter la définition de `CONFIG_TEXTUI` dans les options de construction du programme pour activer correctement la compilation conditionnelle.

2.2.5 Validation de la logique de jeu et bouchonnage de *player_manager*

Proposez un jeu de tests pour valider la logique du jeu.

Écrivez la fonction `PlayerManager_oneTurn` du module *player_manager_mock* pour jouer automatiquement le jeu de tests que vous avez défini.

Remarque pour la construction

Activez la définition de `CONFIG_PLAYER_MANAGER_MOCK` dans les options de construction du programme pour activer correctement la compilation conditionnelle.

Le schéma 4 vous aidera à coder le corps de cette fonction.

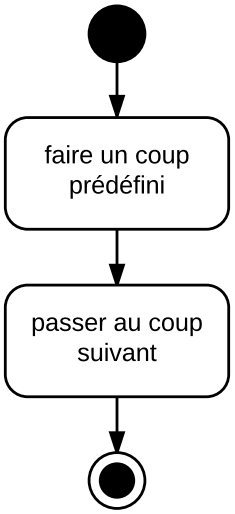

FIGURE 4 – Algorithme de `oneTurn`

2.2.6 Module *Game*

Écrivez le corps des fonctions `Game_init`, `Game_free` et `Game_loop`. L'algorithme de cette dernière est schématisé dans la figure 5 page suivante en collaboration avec d'autres

fonctions et notamment les fonctions « callback » à écrire (`SquareChangeCallback` et `EndOfGameCallback`) et dont il faudra fournir l'adresse à la fonction `Board_init`.

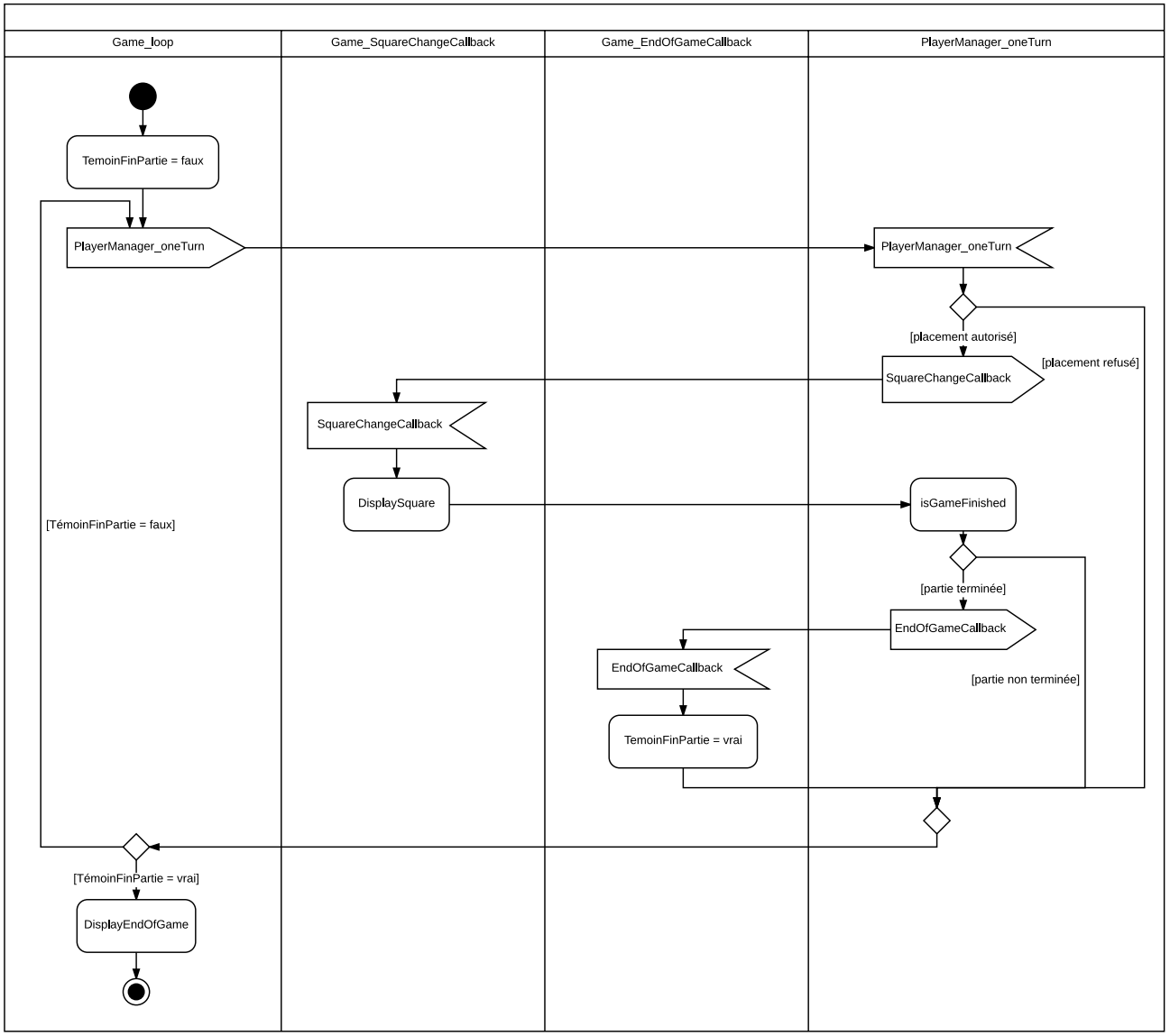

FIGURE 5 – Algorithme de `Game_loop`

2.3 Récapitulatif des actions à réaliser

Vérifiez que vous avez bien réalisé chacune des actions de cette étape.

Action	Fait
codage de `isGameFinished`	
tests de la fonction `isGameFinished`	
codage de `Board_putPiece`	
appel des « callbacks » `SquareChangeCallback` et `EndOfGameCallback` dans la fonction `Board_putPiece`	
codage de `Board_squareContent`	
codage de `Board_init`	
codage de `Board_free`	
codage des fonctions du fichier *board_view_text.c*	
codage des fonctions du fichier *player_manager_mock.c*	
validation de cette mouture du jeu de Morpion	

3 Étape 2

3.1 Objectifs généraux

À présent le moteur de jeu correctement réalisé, vous allez le rendre interactif. Dans un premier temps, pour nous concentrer sur une conception modulaire et une réalisation progressive, il s'agit de rester sur une interface en mode texte. L'utilisateur devra donc saisir les coordonnées de la case où il souhaite jouer.

Il faudra bien évidemment effectuer une validation de votre travail avant de passer à l'étape suivante.

3.2 Description du travail à réaliser

Vous trouverez la base du code source de l'étape 2 (à ajouter à votre projet) sur le campus.

Remarque pour la construction

Après avoir intégré les fichiers, il faut adapter les paramètres de construction de manière à définir `CONFIG_PLAYER_MANAGER_SCANF` à la place de `CONFIG_PLAYER_MANAGER_MOCK`.

3.2.1 Fonction `PlayerManager_oneTurn`

Cette fonction, lorsqu'elle est appelée, doit réaliser un tour de jeu. Cela implique donc qu'elle soit à l'écoute des actions de l'utilisateur.

Dans le cas présent, l'utilisateur devra saisir les coordonnées de la case en respectant les définitions suivantes :

- le format de saisie est rigoureusement « `X,Y` » ;
- il n'y a notamment pas d'espace ni d'autre séparateur que la virgule entre les deux coordonnées ;
- X est le numéro de colonne dans le tableau ;
- Y est le numéro de la ligne dans le tableau ;
- les numéros (de ligne ou de colonne) sont compris entre 0 et 2 inclus.

À partir des coordonnées saisies, il faut utiliser le module *board* afin de placer effectivement un pion dans la case correspondante.

En cas de saisie incorrecte (non respect des définitions au dessus ou placement sur une case non vide), la fonction devra attendre une nouvelle saisie de l'utilisateur.

3.2.2 Validation

Définissez un jeu de données montrant que votre programme, avec la saisie utilisateur :

- respecte toujours le cahier des charges (pas de régression sur le fonctionnement de la logique de jeu) ;
- fonctionne pour toutes les saisies utilisateurs nominales (auxquelles on peut s'attendre) ;
- ne dysfonctionne pas en cas de saisie non nominale (robustesse).

3.3 Récapitulatif des actions à réaliser

Vérifiez que vous avez bien réalisé chacune des actions de cette étape.

Action	**Fait**
paramétrage de la compilation conditionnelle	
codage de la fonction `PlayerManager_oneTurn`	
validation de la saisie utilisateur	

4 Étape 3

4.1 Objectifs généraux

Nous souhaitons à présent donner un peu d'allure à notre jeu en utilisant une bibliothèque permettant de construire une interface graphique : la bibliothèque SDL2 [2].

Une capture d'écran du jeu terminé est présentée en figure 6 page suivante. Elle est basée sur des images (fournies sur le campus, avec les fichiers de l'étape 3) que vous pourrez adapter à votre goût.

Pour gagner du temps sur l'exercice, l'essentiel du code spécifique à l'utilisation de la bibliothèque SDL est déjà fourni. Il reste cependant des parties très spécifiques à l'application courante, telles que les dimensions graphiques.

2. *http://www.libsdl.org/*

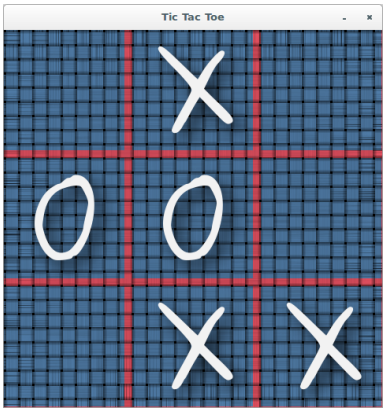

FIGURE 6 – Capture d'écran du jeu terminé

Une fois complétée, il faudra bien sûr valider cette interface.

4.2 Description du travail à réaliser

Vous trouverez la base du code source de l'étape 3 (à ajouter à votre projet) sur le campus.

Nous allons, dans un premier temps, réaliser le module *board_view_sdl* avant de prendre en compte les interactions utilisateur dans le module *square_manager_sdl*.

4.2.1 Utilisation du module *board_view_sdl*

En vous inspirant de ce qui a été fait précédemment et du code source, activez la compilation conditionnelle pour le module *board_view_sdl*.

4.2.2 Fonction `BoardView_displayAll`

En utilisant la fonction privée `renderImage`, affichez l'image de fond préchargée dans la variable globale `BackgroundImage` [3].

Ensuite, en appelant la fonction `BoardView_displaySquare`, affichez chacune des cases du tableau. Bien sûr, cet appel restera sans effet tant que vous n'aurez pas codé cette même fonction ; mais cela viendra dans un second temps, lorsque vous aurez réussi à afficher l'image de fond.

Validez le bon fonctionnement de l'affichage du fond en exécutant le programme.

4.2.3 Fonction `BoardView_displaySquare`

En utilisant la spécification des dimensions donnée en figure 7 page suivante et la fonction `renderImage`, écrivez le corps de la fonction `BoardView_displaySquare` pour placer la

3. Vous pouvez réadapter le code de chargement dans les lignes de la fonction `BoardView_init`

bonne image (chargées dans les variables `SpriteO` et `SpriteX`) au bon endroit dans la fenêtre, par rapport aux coordonnées logiques fournies.

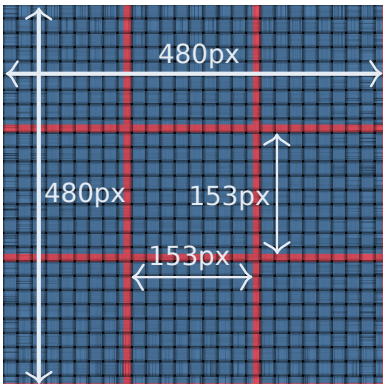

FIGURE 7 – Dimensions des images du jeu

Validez le bon fonctionnement :

- l'image s'affiche dans la bonne case ;
- l'image est la bonne ;
- l'image est positionnée exactement (au pixel près).

4.2.4 Utilisation du module *player_manager_sdl*

Activez la compilation conditionnelle pour le module *player_manager_sdl*.

4.2.5 Fermeture de la fenêtre

Dans un premier temps, pour des raisons pratiques, nous allons faire en sorte de terminer l'application lors d'une demande de fermeture de la fenêtre.

Pour rappel, la fonction `PlayerManager_oneTurn` attend la saisie de l'utilisateur (essentiellement à travers `SDL_waitEvent`.

Dans le cas de notre application, nous pouvons résumer le type d'événement `SDL_WINDOWEVENT` à une demande de fermeture de fenêtre. Adaptez le cas de figure correspondant pour quitter l'application (voir fonction `exit`).

4.2.6 Prise en compte du clic souris

En vous aidant de la documentation de la bibliothèque SDL 2.0[4] et plus particulièrement de ce qui concerne le type `SDL_EvenType`, créez un nouveau cas de figure dans la fonction `PlayerManager_oneTurn` pour prendre en compte un clic de souris.

4. Voir le site web *http://www.libsdl.org*

Appuyez-vous enfin sur l'algorithme représenté en figure 8 pour terminer l'écriture de la fonction `PlayerManager_oneTurn`.

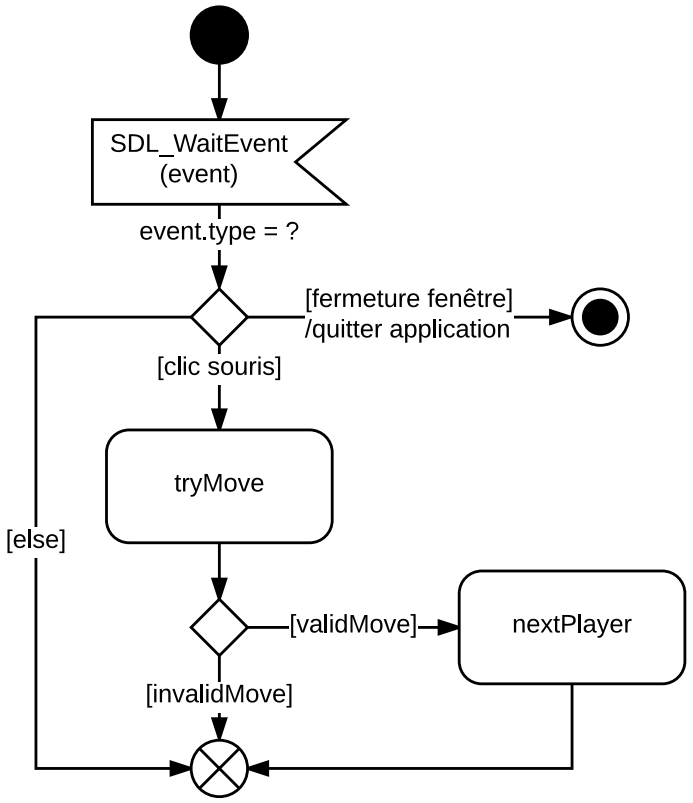

FIGURE 8 – Algorithme de la fonction `PlayerManager_oneTurn`

4.2.7 Fonction `tryMove`

Pour coder cette fonction, il s'agit surtout de convertir correctement des coordonnées du clic dans la fenêtre en coordonnées logiques d'une case dans le tableau.

Une fois la conversion effectuée, si le clic est dans une case, il faut appeler la fonction `Board_putPiece` pour placer effectivement un pion. En cas de placement de pion invalide, il faut appeler la fonction `BoardView_sayCannotPutPiece` pour avertir l'utilisateur.

4.2.8 Validation

Validez le fonctionnement de votre application :

- comportement lors du clic à l'intérieur, proche du centre d'une case ;
- comportement lors du clic à l'intérieur, proche de l'extérieur d'une case ;
- comportement lors du clic dans l'espace entre des cases.

4.3 Récapitulatif des actions à réaliser

Vérifiez que vous avez bien réalisé chacune des actions de cette étape.

Action	**Fait**
paramétrage de la compilation conditionnelle pour le module *board_view_sdl*	
codage de la fonction `BoardView_displayAll`	
codage de la fonction `BoardView_displaySquare`	
paramétrage de la compilation conditionnelle pour le module *player_manager_sdl*	
gestion de la fermeture de la fenêtre	
interprétation du clic souris	
action conséquente au clic souris	
validation générale du jeu	

5 Aller plus loin…

Vous pouvez améliorer les aspects graphiques du jeu, notamment implémenter les fonctions comme `BoardView_displayPlayersTurn` en SDL. Mais vous devriez désormais être prêts à réaliser votre propre jeu (ou autre application similaire) en vous inspirant de ce que vous avez fait ici.

Autrement dit, passez tout de suite à la réalisation de votre « projet libre » en commençant, si ce n'est pas déjà fait, par faire valider votre sujet par un enseignant.