ECOLE SUPERIEURE D'ELECTRONIQUE DE L'OUEST (ESEO)

Livrable de : PDLO

Nom et Prénom :

Ornel ZINSOU-PLY

Extension

Extension 1 :
Ajouter les pièces manquantes

- Dans un ordre indiqué
- Ne pas oublier de modifier l'UsineDePiece()

***Extension 2* :**
Détecter quand une ligne est complète dans le tas, pour la supprimer (et faire descendre les pièces plus haut - pour réduire la hauteur du tas)

***Extension 3* :**
Utiliser le clavier pour la rotation et le mouvement des pièces

***Extension 4* :**
Ajouter un bouton qui permettre de mettre en pause une partie du jeu

Extension 1 :

- Ajouter les pièces manquantes (voir figure 1)
  - dans l'ordre suivant:
    - `TTetromino`,
    - `LTetromino`,
    - `JTetromino`,
    - `ZTetromino`,
    - `STetromino`)
  - Ne pas oublier de modifier l'`UsineDePiece` pour que ces nouvelles Pieces puissent être généré. (Le `MODE_CYCLIC` doit respecter l'ordre ci-dessus).

Ajouter les pièces manquantes

Il consiste à la création des classes : TTetromino.java, LTetromino.java, JTetromino.java, ZTetromino.java et STetromino.java

Chacune de ces classes, héritent de la classe ***Tetromino.java*** d'où le mot ***extends*** .

TTetromino.java

```
public class TTetromino extends Tetromino {

public TTetromino(Coordonnees coordonnees, Couleur couleur) {
                super(coordonnees, couleur);
        }

protected void setElements(Coordonnees coordonnees, Couleur couleur){

// code sur le fonctionnement du setElements

}

}
```

LTetromino.java

```
public class LTetromino extends Tetromino {

public LTetromino(Coordonnees coordonnees, Couleur couleur) {
                super(coordonnees, couleur);
        }

protected void setElements(Coordonnees coordonnees, Couleur couleur){

// code sur le fonctionnement du setElements

}
```

```
}
```

JTetromino.java

```
public class JTetromino extends Tetromino {

public LTetromino(Coordonnees coordonnees, Couleur couleur) {
        super(coordonnees, couleur);
    }

protected void setElements(Coordonnees coordonnees, Couleur couleur){

// code sur le fonctionnement du setElements

}

}
```

ZTetromino.java

```
public class ZTetromino extends Tetromino {

public JTetromino(Coordonnees coordonnees, Couleur couleur) {
        super(coordonnees, couleur);
    }

protected void setElements(Coordonnees coordonnees, Couleur couleur){

// code sur le fonctionnement du setElements

}

}
```

STetromino.java

```
public class STetromino extends Tetromino {

public ZTetromino(Coordonnees coordonnees, Couleur couleur) {
        super(coordonnees, couleur);
    }

protected void setElements(Coordonnees coordonnees, Couleur couleur){

// code sur le fonctionnement du setElements

}


}
```

Au sein de chacune de ces classes, la méthode :

```
 protected void setElements(Coordonnees coordonnees, Couleur couleur){
```

} est écrite en tenant compte du schéma ci-dessous.

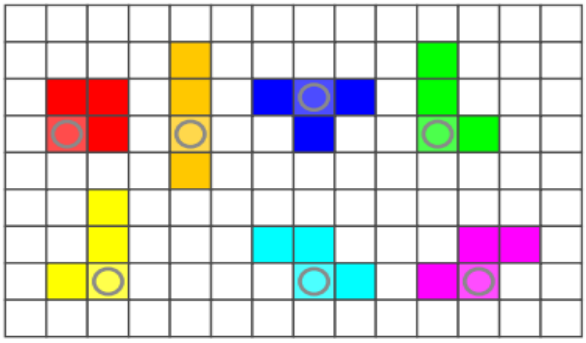
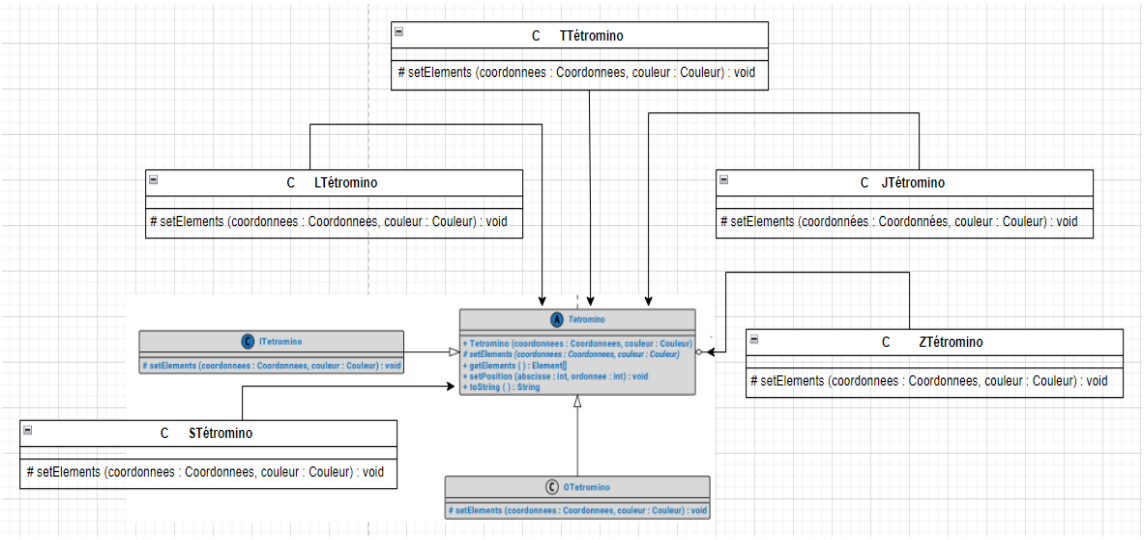


Figure 1: Les différentes Tetrominos: OTetromino (rouge), ITetromino (orange), TTetromino (bleu), LTetromino (vert), JTetromino (jaune), ZTetromino (cyan), STetromino (violet)

Diagramme UML illustrant le code fait au-dessus :



Ne pas oublier de modifier l'UsineDePiece pour que ces nouvelles Pieces puissent être généré. (Le MODE_CYCLIC doit respecter l'ordre ci-dessus).

Dans UsineDePiece, se trouve la méthode **public static** Tetromino genererTetrominoCyclique(). Avec cette méthode je génère le mode cyclique dans l'ordre des tétrominos indiqués.

```java
public static Tetromino genererTetrominoCyclique() {
    Tetromino tetromino = null;
    if (modeDeFonctionnement == 3) { // CYCLIC
        if (j == 0) {
            tetromino = new OTetromino(new Coordonnees(2, 3), Couleur.ROUGE);
            j = 1;
        } else if (j == 1) {
            tetromino = new ITetromino(new Coordonnees(2, 3), Couleur.ORANGE);
            j = 2;
        } else if (j == 2) {
            tetromino = new TTetromino(new Coordonnees(2, 3), Couleur.BLEU);
            j = 3;
        } else if (j == 3) {
            tetromino = new LTetromino(new Coordonnees(2, 3), Couleur.VERT);
            j = 4;
        } else if (j == 4) {
            tetromino = new JTetromino(new Coordonnees(2, 3), Couleur.JAUNE);
            j = 5;
        } else if (j == 5) {
            tetromino = new ZTetromino(new Coordonnees(2, 3), Couleur.CYAN);
            j = 6;
        } else if (j == 6) {
            tetromino = new STetromino(new Coordonnees(2, 3), Couleur.VIOLET);
            j = 0;
        }
    }
    return tetromino;
}
```

Extension 2 :

Détecter quand une ligne est complète dans le tas, pour la supprimer (et faire descendre les pièces plus haut - pour réduire la hauteur du tas)

```java
public void verifierEtNettoyerLignes() {
    int profondeur = puits.getProfondeur();
    int largeur = puits.getLargeur();
    boolean lineComplete;

    for (int y = 0; y < profondeur; y++) {
        lineComplete = true;
        for (int x = 0; x < largeur; x++) {
            if (!elementExists(x, y)) {
                lineComplete = false;
                break;
            }
        }
        if (lineComplete) {
            supprimeligne(y);
            descendElements(y);
            y--;
        }
    }
}

private void supprimeligne(int y) {
    for (int i = 0; i < elements.size(); i++) {
        if (elements.get(i).getCoordonnees().getOrdonnee() == y) {
            elements.remove(i);
            i--;
            puits.setScore(puits.getScore() + 10 * lignesupprimé);
        }
    }
}

private void descendElements(int y) {
    for (int i = 0; i < elements.size(); i++) {
        Element element = elements.get(i);
        if (element.getCoordonnees().getOrdonnee() < y) {
            element.getCoordonnees().setOrdonnee(element.getCoordonnees().getOrdonnee() + 1);
        }
    }
}
```

Le code se compose de 3 parties :

Méthode ***verifierEtNettoyerLignes()*** :

Cette méthode parcourt toutes les lignes du jeu de haut en bas pour vérifier si elles sont complètes, c'est-à-dire si chaque colonne de la ligne contient un élément.

Si une ligne complète est trouvée, elle est supprimée et tous les éléments au-dessus descendent d'une position.

Méthode ***suppriméligne(int y)*** :

Cette méthode supprime tous les éléments d'une ligne donnée (y est l'index de la ligne).

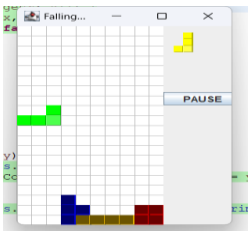
Elle parcourt également tous les éléments du jeu, et chaque fois qu'un élément de la ligne à supprimer est trouvé, il est retiré de la liste des éléments.

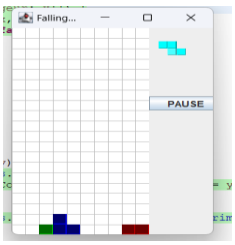
Le score du jeu est augmenté chaque fois qu'une ligne est supprimée.

Méthode ***descendElements(int y)*** :

Après qu'une ligne est supprimée, cette méthode est appelée pour déplacer tous les éléments qui étaient au-dessus de cette ligne vers le bas, en incrémentant leur position en ordonnée (verticale).

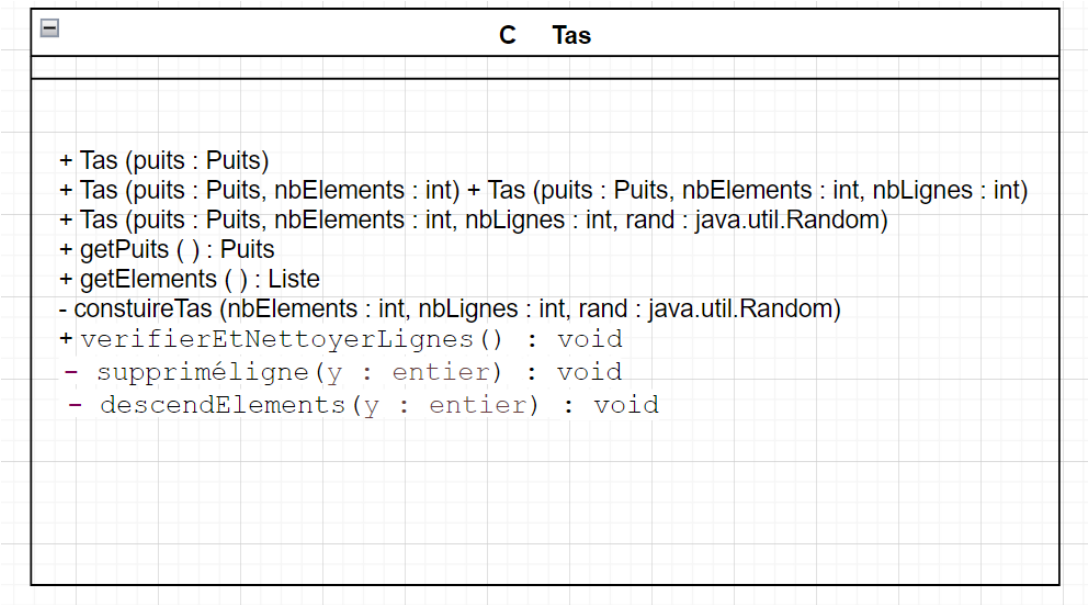
Résultat avant tombé du LTetromino (vert) **Résultat après tombé du LTetromino (vert)**





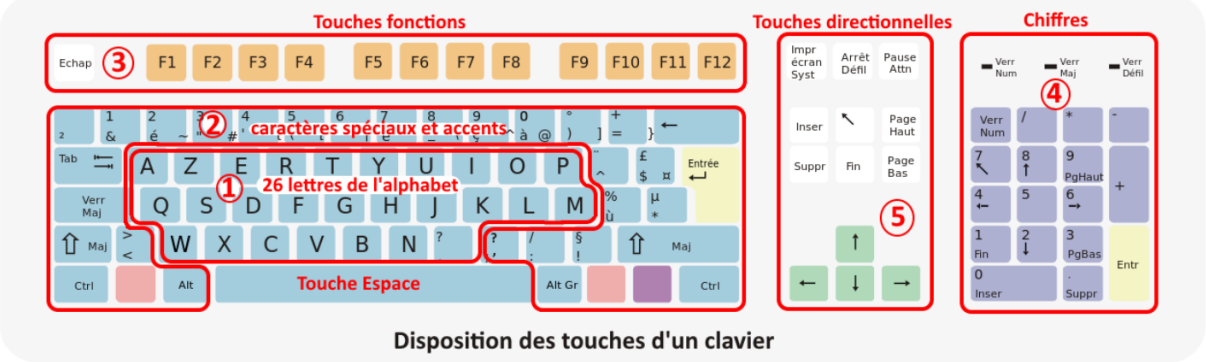
On observe réellement un changement du tas au fond du puits.

Diagramme UML illustrant le code fait au-dessus :



Extension 3 :

Utiliser le clavier pour la rotation et le mouvement des pièces



Un clavier est composé majoritairement de 5 parties. Pour la rotation et le mouvement des pièces, je me suis concentré que sur deux parties du clavier à savoir :

- 1) 26 lettres de l'alphabet
- 2) Touche directionnelles

Rotation de la pièce

Au niveau des 26 lettres de l'alphabet du clavier, on a la première lettre de l'alphabet qui est « A » . Lorsque la ***Touche 'A'*** qui contient cette lettre est enfoncé, la pièce actuelle situé dans le puit est tourné dans le sens anti-horaire.

Condition: *if (e.getKeyCode() == KeyEvent.VK_A) {...}* vérifie si la touche pressée est la touche 'A'.

Action:

vuePuits.getPuits().getPieceActuelle().tourner(false); est appelé pour tourner la pièce actuelle dans une direction (ici représentée par false). La méthode tourner semble un booléen ***senshoraire*** pour déterminer la direction de rotation.

Gestion des exceptions: Si la rotation échoue pour des raisons de validation (Illégale) ou d'autres problèmes spécifiques au jeu (BloxException), une nouvelle exception est levée avec un message explicatif.

Ensuite, au niveau des touches directionnelles nous avons cette ***Touche 'UP' (flèche vers le haut)***

Cette dernière quand elle est appuyée, elle permet de tourner la pièce actuelle situé dans le puit dans le sens horaire.

Condition externe*: if (vuePuits.getPuits().getPieceActuelle() != null) {...}* assure que la pièce actuelle n'est pas null (c'est-à-dire qu'il y a une pièce présente à manipuler).

Condition interne*: if (e.getKeyCode() == KeyEvent.VK_UP) {...}* vérifie si la touche pressée est la flèche vers le haut.

Action:

vuePuits.getPuits().getPieceActuelle().tourner(true); tente de tourner la pièce dans l'autre direction (ici true).

Gestion des exceptions: Similaire à la première condition, gère les exceptions et renvoie un message si une erreur survient.

Ce qui se traduit par le code ci-dessous :

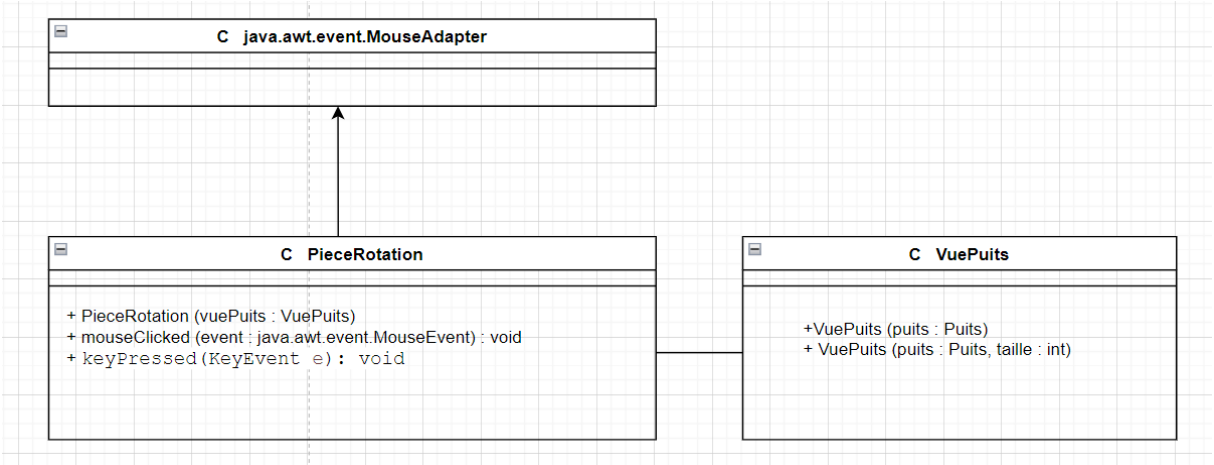
```
@Override
public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_A) {
        try {
            vuePuits.getPuits().getPieceActuelle().tourner(false);
        } catch (IllegalArgumentException | BloxException e1) {
            throw new IllegalArgumentException("Impossible de tourner la piece." + e1.getMessage());
        } finally {
            vuePuits.repaint();
        }
    }

    if (vuePuits.getPuits().getPieceActuelle() != null) {
        if (e.getKeyCode() == KeyEvent.VK_UP) {
            try {
                vuePuits.getPuits().getPieceActuelle().tourner(true);
            } catch (IllegalArgumentException | BloxException e1) {
                throw new IllegalArgumentException("Impossible de tourner la piece." + e1.getMessage());
            } finally {
                vuePuits.repaint();
            }
        }
    }
}
```

Bloc 'finally'

Dans chaque bloc *try*, il y a un bloc *finally* qui s'exécute indépendamment du fait que le *try* ait réussi ou échoué.

vuePuits.repaint(); dans le bloc *finally* est appelé pour redessiner ou rafraîchir la vue graphique *(vuePuits).* Ceci est nécessaire pour mettre à jour l'interface graphique avec la nouvelle position ou orientation de la pièce.

Diagramme UML illustrant le code fait au-dessus :



Mouvement de la pièce

Nous avons une condition initiale: ***if (vuePuits.getPuits().getPieceActuelle() != null) {...}*** s'assure qu'il y a une pièce actuelle à manipuler, c'est-à-dire que la pièce actuelle n'est pas nulle.

Pour la gestion des touches, on a :

***Flèche Gauche** (**KeyEvent.VK_LEFT**):*

vuePuits.getPuits().getPieceActuelle().deplacerDe(-1, 0); essaie de déplacer la pièce d'une unité vers la gauche sur la grille. Le premier argument **-1** signifie "déplacer d'un pas à gauche", et le **0** signifie "ne pas déplacer verticalement".

***Flèche Droite** (**KeyEvent.VK_RIGHT**):*

vuePuits.getPuits().getPieceActuelle().deplacerDe(1, 0); essaie de déplacer la pièce d'une unité vers la droite. Ici, **1** indique "un pas à droite".

***Flèche Bas** (**KeyEvent.VK_DOWN**):*

vuePuits.getPuits().getPieceActuelle().deplacerDe(0, 1); essaie de déplacer la pièce d'une unité vers le bas. Le **0** est pour le déplacement horizontal, et **1** pour un pas vers le bas.

Gestion des Exceptions

Chaque opération de déplacement est enveloppée dans un bloc *try-catch*. Si une exception *IllegalArgumentException* ou *BloxException* est levée pendant le déplacement, le bloc **catch** est prêt à les intercepter. Cependant, aucun traitement spécifique n'est fait dans ces blocs **catch**, ils sont vides. Cela signifie que le programme ignore silencieusement l'erreur et continue.

Bloc finally

Après chaque tentative de déplacement, indépendamment du succès ou de l'échec, le bloc *finally* est exécuté.

vuePuits.repaint(); est appelé pour redessiner la vue graphique, ce qui est essentiel pour que les changements de position de la pièce soient visibles sur l'écran.

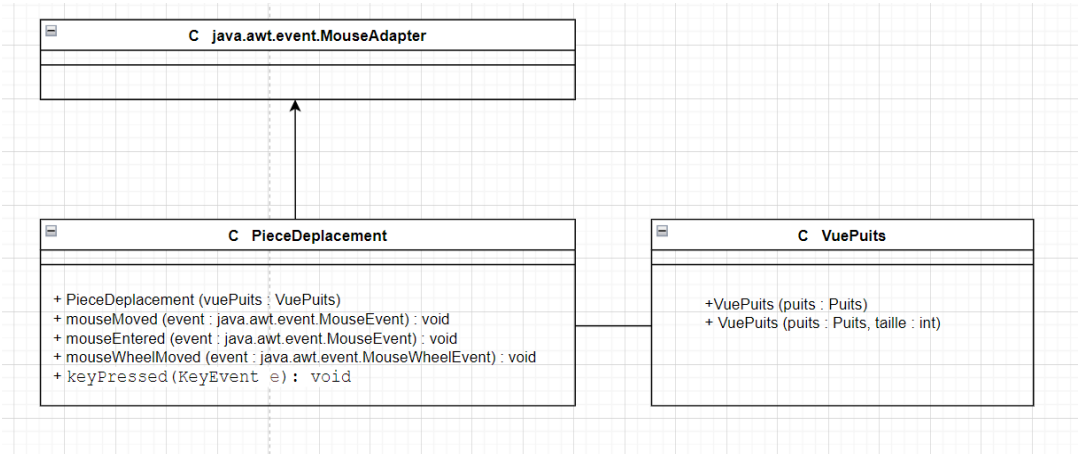
Clause else

else { return; } est exécuté si aucune pièce n'est actuellement sélectionnée (**getPieceActuelle()** retourne **null**). Cela termine l'exécution de la méthode sans faire d'autres actions.

Ce qui se traduit par le code ci-dessous :

```java
@Override
public void keyPressed(KeyEvent e) {
    // TODO Auto-generated method stub
    if (vuePuits.getPuits().getPieceActuelle() != null) {
        if (e.getKeyCode() == KeyEvent.VK_LEFT) {
            try {
                vuePuits.getPuits().getPieceActuelle().deplacerDe(-1, 0);
            } catch (IllegalArgumentException | BloxException e1) {

            } finally {
                vuePuits.repaint();
            }
        } else if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
            try {
                vuePuits.getPuits().getPieceActuelle().deplacerDe(1, 0);
            } catch (IllegalArgumentException | BloxException e1) {

            } finally {
                vuePuits.repaint();
            }
        } else if (e.getKeyCode() == KeyEvent.VK_DOWN) {
            try {
                vuePuits.getPuits().getPieceActuelle().deplacerDe(0, 1);
            } catch (IllegalArgumentException | BloxException e1) {

            } finally {
                vuePuits.repaint();
            }
        }
    } else {
        return;
    }
}
```

Diagramme UML illustrant le code fait au-dessus :



<u>Extension 4</u> :

Ajouter un bouton qui permettre de mettre en pause une partie du jeu

On a la méthode *propBouton()*. Cette méthode initialise et configure un bouton, utilisé pour mettre le jeu en pause ou le reprendre.

button = new JButton(); : Crée un nouvel objet bouton.

this.button.setText("PAUSE"); : Définit le texte initial du bouton à "PAUSE".

this.button.setBounds(0, 100, 80, 20); : Définit la position et la taille du bouton dans la fenêtre. Ici, le bouton est placé à l'abscisse 0, l'ordonnée 100, avec une largeur de 80 et une hauteur de 20 pixels.

this.button.addActionListener(this); : Enregistre l'objet courant (*this*) comme écouteur des actions réalisées sur le bouton. Cela signifie que *this* implémente l'interface *ActionListener*, et la méthode *actionPerformed* sera appelée lorsque le bouton est cliqué.

this.add(button); : Ajoute le bouton à la fenêtre ou au panneau conteneur.

repaint(); : Demande à l'interface graphique de se redessiner. Ceci est utile pour s'assurer que le bouton apparaît correctement sur l'écran.

Le fonctionnement de ce bouton est l'œuvre de plusieurs autres métodes.

La méthode *actionPerformed(ActionEvent e)*

Cette méthode est appelée automatiquement lorsque l'utilisateur interagit avec les composants graphiques qui ont cet objet comme écouteur.

if (e.getSource() == this.button) { }: Vérifie si l'action provient du bouton.

puits.setPaused(!puits.getIsPause()); : Change l'état de pause du jeu. Si le jeu est en pause, il le reprend, et vice versa.

Si le jeu est maintenant en pause (*if (puits.getIsPause())*), change le texte du bouton en ***"PLAY".*** Sinon, remet le texte à ***"PAUSE"***.

La méthode gravite()

Cette méthode gére le déplacement automatique des pièces vers le bas, simulant la gravité.

if (!mettrePause) { }: Vérifie si le jeu n'est pas en pause.

getPieceActuelle().deplacerDe(0, 1); : Tente de déplacer la pièce actuelle d'une position vers le bas. Le (0, 1) signifie qu'il n'y a pas de déplacement horizontal, mais un déplacement vertical d'une unité.

catch (BloxException e) {} : Attrape les exceptions spécifiques au jeu.

if (e.getType() == BloxException.BLOX_COLLISION) { }: Si l'exception est due à une collision, appelle *gererCollision();* pour gérer la situation.

Méthodes *setPause(boolean pause)* et *getIsPause()*

Ces deux méthodes gèrent l'état de pause du jeu.

setPause(boolean pause) : Permet de définir l'état de pause.

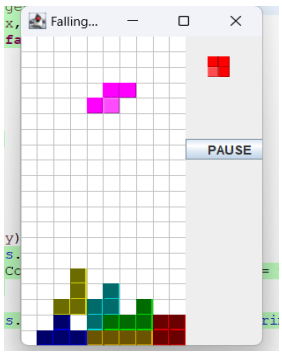
getIsPause() : Retourne l'état de pause actuel.

Ce qui se traduit par le code ci-dessous :

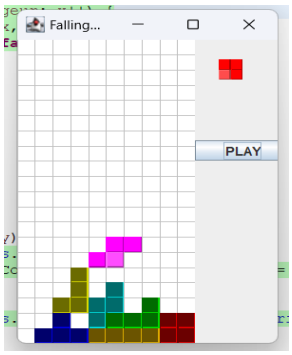
```java
// pause --> play
private void propBouton() {
    button = new JButton();
    this.button.setText("PAUSE");
    this.button.setBounds(0, 100, 80, 20);
    this.button.addActionListener(this);
    this.add(button);
    repaint();
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == this.button) {
        puits.setPause(!puits.getIsPause());
        if (puits.getIsPause()) {
            button.setText("PLAY");
        } else {
            button.setText("PAUSE");
        }
    }
}
```

```java
public void gravite() {
    if (!mettrePause) {
        try {
            getPieceActuelle().deplacerDe(0, 1);
        } catch (BloxException e) {
            if (e.getType() == BloxException.BLOX_COLLISION) {
                gererCollision();
            }
        }
    }
}

public void setPause(boolean pause) {
    this.mettrePause = pause;
}

public boolean getIsPause() {
    return mettrePause;
}
```

Résultat à la PAUSE (le jeu s'arrête)



Résultat pendant PLAY (la pièce descend)



On observe réellement que le bouton PAUSE → PLAY fonctionne.

Diagramme UML illustrant le code fait au-dessus :