GRANDE ÉCOLE D'INGÉNIEURS

Falling Blox

Richard WOODWARD

April 15, 2024

Important!

Les informations dans le présent document sont susceptibles d'être modifiée. Il est conseillé de ne pas ni télécharger, ni imprimer ce document et d'utiliser toujours la version en ligne.

Contents

1 Introduction

1.1 Introduction Générale

Ce document est l'énoncé du projet associé à l'ECUE Programmation Orientée Objets en Java. L'objectif de ce travail est de vous faire pratiquer, à travers un exemple concret, les concepts de la programmation orientée objets et de la programmation en java. Ainsi, le développement de l'application décrit dans ce document vous permettra de manipuler:

- les classes, interfaces et énumérations en Java
- les paquetages en Java
- les exceptions standards et les exceptions personnalisées
- les interfaces hommes-machines s'appuyant sur les composants et évènements Swing
- ...

La version de Java qui est conseillé pour ce projet est la version 11.

1.2 L'Application

Dans le cadre de ce projet,l'application qui vous demande de construire ressemble beaucoup au fameux jeu qui a été développé par l'informaticien russe Alexi Pajitonov: Tetris. En suivant les instructions données dans ce document, vous pourrez créer une première version minimale -mais fonctionnelle- du jeu. Lorsque vous aurez terminé le travail qui vous est demandé dans ce document, vous serez libre d'ajouter les fonctionnalités manquantes à cette première version, pour en faire une implémentation complète de Tetris en java.

1.2.1 Les règles du jeu

Les règles de Tetris sont très simples. Le joueur se voit présenter un puits dans lequel une pièce (qui peut être soit un tétromino - composé de quatre éléments - soit un pentomino - composé de cinq éléments) choisie au hasard parmi les différentes formes disponibles (voir la figure 1 pour les tétrominos possibles), commence à tomber sous l'effet de la gravité.

Lors de sa chute, le joueur peut manipuler la pièce, soit en la déplaçant vers la gauche ou la droite, soit en tournant la pièce de 90° autour d'un élément donné (sur la figure 1, ce sont les éléments mis en évidence) soit dans le sens des aiguilles d'une montre, soit dans le sens inverse.

Une fois qu'une pièce arrive au fond du puits, la pièce cesse de bouger et les éléments individuels qui la composent s'ajoutent à une pile d'éléments. Les éléments d'une pièce seront également ajoutés à la pile, s'ils entrent en contact avec les éléments déjà présents dans la pile.

Lorsque les éléments de la pièce sont ajoutés à la pile, s'ils créent une ligne horizontale traversant le puits, les éléments de cette ligne sont supprimés, le joueur marque quelques points et les éléments de la pile situés au-dessus de la ligne supprimée tombent vers le bas.

Si la pile arrive au sommet du puits, la partie est terminée.

Dans certaines versions, les pièces qui tombent sont choisies parmi les tétrominos disponibles, sauf lorsque le joueur atteint un certain seuil de points. À ce stade, la vitesse du jeu augmente, et un pentomino est sélectionné comme prochaine pièce à tomber dans le puits.

1.2.2 Notre implémentation

Comme déjà écrit, ce projet vous permettra dans un premier temps, de créer une version simplifiée de Tetris, dans laquelle deux types de Piece (les O-Tétrominos et I-Tétrominos) tombent dans le puits. Le joueur utilisera la souris pour contrôler la pièce. La pile sera construite en fonction des pièces qui tomberont.

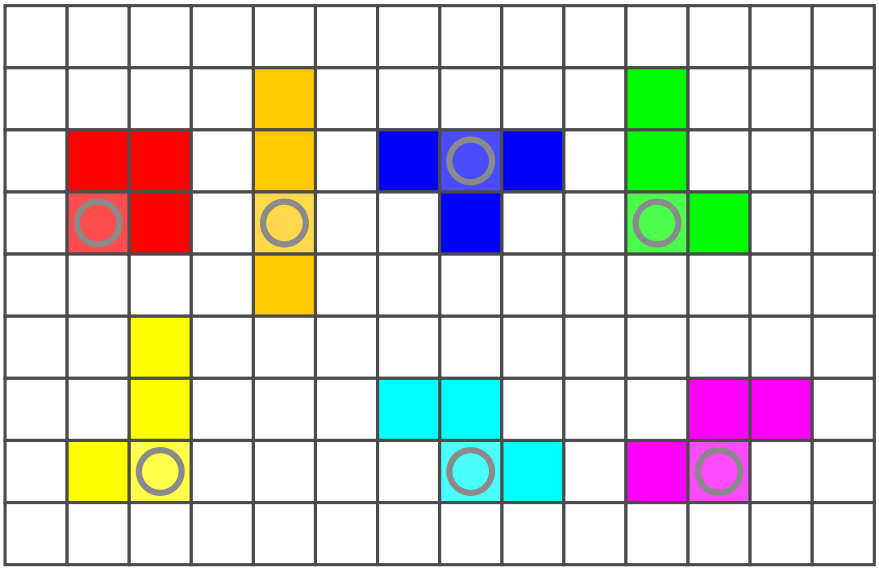


Figure 1: Les différentes Tetrominos: OTetromino (rouge), ITetromino (orange), TTetromino (bleu), LTetromino (vert), JTetromino (jaune), ZTetromino (cyan), STetromino (violet)

Dans un deuxième temps (une fois les instructions de la section 3 terminées), un certain nombre d'extensions peuvent être ajoutées pour compléter la fonctionnalité. La section 4 répertorie les extensions possibles qui doivent ou pourraient être mises en œuvre.

1.3 Le schéma de conception MVC

(inspiré de http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamental) - lien mort)

Le patron d'architecture Modèle-Vue-Contrôleur (MVC) est assez ancien. Il est apparu aux environs des premiers jours de Smalltalk. C'est un patron de haut niveau dans le sens où il s'occupe de l'architecture globale d'un programme et essaie de fournir une classification des différents types d'objets qui composent une application.

D'après cette architecture, il y a trois types d'objets : les objets Modèle, les objets Vue et les objets Contrôleur.

L'architecture définit les rôles que jouent les 3 types d'objets dans l'application. En tant que développeur, vous créez vos classes en fonction de ces trois groupes d'objets. Chacun d'eux est séparé des autres par des frontières abstraites, et communique avec les objets d'un autre type à travers ces frontières.

1.3.1 Les objets "Modèle" représentent les données et les fonctionnalités de base

Les objets Modèle représentent les connaissances et expertises spécifiques à l'application. Ils contiennent les données d'une application et définissent la logique de manipulation de ces données. Une application construite correctement sur le modèle MVC a ses données les plus importantes encapsulées dans des objets modèle. Toute donnée qui fait partie de l'état persistant d'une application (les réglages par défaut par exemple) (que cet état soit stocké dans des fichiers, dans des bases de données ou dans des cartes perforées) devrait être incorporée dans les objets Modèle une fois la donnée chargée dans l'application.

Idéalement, un objet Modèle n'a pas de connexion avec l'interface utilisateur permettant de le visualiser et de le modifier. Dans notre application d'éditeur graphique, les formes géométriques seront des objets Modèle et n'auront aucun besoin, dans un premier temps, de savoir comment ils sont présentés à l'utilisateur.

En pratique, la séparation nette entre objet Modèle et interface n'est pas toujours souhaitable, et il y a des marges de manœuvre. Cependant, en général, un objet Modèle ne doit pas être concerné par des considérations d'interface ou de présentation. Ils seront associés à des objets Vue qui seront, eux, chargés de l'affichage des objets modèle.

1.3.2 Les objets "Vue" présentent les données à l'utilisateur

Un objet Vue sait comment afficher et éventuellement modifier les données du modèle d'application. En revanche, il n'est pas responsable du stockage des données qu'il affiche. (cela ne signifie pas que l'objet Vue ne stocke jamais les données qu'il affiche; un objet Vue peut mettre en cache les données ou faire d'autres choses similaires pour des raisons de performance).

Un objet Vue peut être chargé d'afficher une partie d'un objet modèle ou un objet modèle entier ou bien encore plusieurs objets modèle différents. Les objets Vue sont de différentes sortes. Par ailleurs, un objet modèle peut être présenté de diverses manières par différents objets Vue.

Les objets Vue tendent à être réutilisables et fournissent une homogénéité entre les applications. SWING définit un grand nombre d'objets Vue.

Un objet Vue assure que l'objet modèle est affiché correctement. En conséquence, il a besoin de connaître les changements que l'objet modèle subit. Les objets modèle n'étant pas liés directement aux objets Vue, ils ont besoin d'une voie générique pour leur signaler des changements dans leur état. Ainsi, ils peuvent envoyer des notifications à leur Vue pour signaler des changements dans leur état qui impliquent une modification de leur présentation par la Vue. Ce mécanisme de communication est généralement fait via la couche Contrôleur de l'application.

1.3.3 Les objets "Contrôleur" lient les objets "Modèle" et les objets "Vue"

Un objet Contrôleur agit comme un intermédiaire entre les objets Vue et modèle de l'application.

Lorsque l'utilisateur agit sur la Vue, avec sa souris, son clavier, en appuyant sur des boutons ou en sélectionnant des éléments dans des menus, il agit sur des objets Vue et ses actions doivent avoir pour conséquence de modifier les objets Modèle, qui à leur tour peuvent modifier la Vue de l'objet modèle associé. La mise en œuvre de ce processus est assurée par les objets Contrôleur.

Les objets Contrôleur représentent ainsi le canal par lequel les objets Vue et modèle communiquent.

En confinant le code spécifique à l'application dans les objets Contrôleur, on rend les objets modèle et Vue plus génériques et réutilisables. Les objets Contrôleur sont souvent les objets les moins réutilisables d'une application, car ils représentent en quelque sorte la manière dont les données sont affichées et dont l'utilisateur peut interagir avec elles. Par exemple, si on considère un programme dont l'objet est de présenter des objets modèle représentant des personnes ; il peut choisir d'imprimer le nom, le prénom et la date de naissance sur la sortie standard ou afficher les caractéristiques d'une personne dans une fenêtre de dialogue. Quelque soit la Vue choisie pour l'application les mêmes objets modèle pourront être réutilisés.

Dans l'exemple ci-dessus, les objets modèle doivent communiquer avec les *Vues* pour les informer des données à afficher. S'il s'agissait de gérer la saisie, par l'utilisateur, des données relatives à une personne, il faudrait alors ajouter un mécanisme permettant aux objets Vue de communiquer avec la couche modèle pour lui transmettre les valeurs saisies par l'utilisateur (dans une liste déroulante, dans un champ texte, depuis un terminal…).

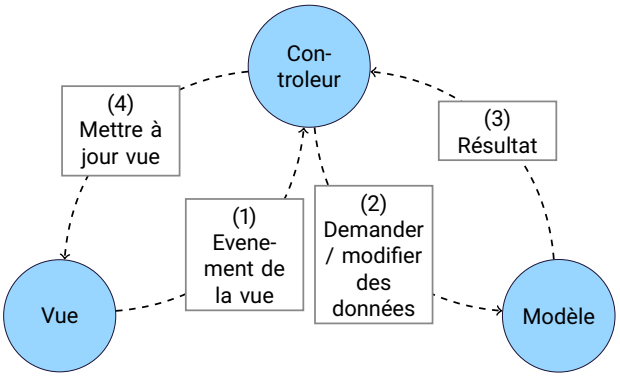


Figure 2: Interactions dans MVC

1.3.4 Le modèle MVC en résumé

Le modèle MVC propose une architecture standard pour la modélisation de toute application graphique. Elle décompose ce type d'application en trois parties distinctes (voir la figure 2) :

- Le modèle qui regroupe les données de l'application (les données métiers);
- la Vue qui permet de proposer des vues des données du modèle ;
- le Contrôleur qui permet à la Vue et au modèle de communiquer l'un avec l'autre.

2 Évaluation du travail

Votre travail sera évalué, au fur et à mesure de son avancement, par un système automatisé : l'ASSIGNMENT CENTRE.

Ce système prend la forme d'une application Java qui se trouve dans l'espace Java du Campus électronique de l'école et qu'il vous sera demandé d'exécuter régulièrement. Il permet de vérifier que les classes écrites par chaque étudiant dans le cadre de ce travail fonctionne correctement et respecte les fonctionnalités décrites dans l'énoncé. Pour que cette application puisse correctement évaluer votre travail, il est extrêmement important que vous utilisiez exactement les même noms et les mêmes visibilités (`public`, `private`, `protected`) de classes, de méthodes et de variables que ceux qui vous sont demandés dans l'énoncé. D'autre part, il est primordial que votre code respecte les formats d'affichage tels qu'ils vous sont demandés (ceux-ci doivent tenir compte de tous les caractères : espaces, marques de ponctuation ...).

Avant de le soumettre à l'ASSIGNMENT CENTRE il est important que vous testiez, vous même, votre code avec des jeux de test de votre composition afin de vous assurer qu'il fonctionne correctement.

Afin de suivre la progression de votre travail, à la fin de chaque séance de Bureau d'Étude, vous devrez charger votre travail sur le campus (sous la forme d'un fichier généré par l'Assignment Centre). Nous vous avertissons que les travaux que vous remettrez sur le campus seront analysés afin de s'assurer qu'il n'y a pas de plagiat et que tout plagiat sera sévèrement puni. Les outils utilisés pour cette détection sont, entre autres, l'outil *Baldr* pour lequel vous trouverez de la documentation à l'URL suivante:

https://github.com/vmarquet/baldr

et JPlag pour lequel vous trouverez de la documentation à l'URL suivante:

https://github.com/jplag/JPlag.

Pour éviter tout soupçonne de plagiat, si une partie de votre code n'est pas votre propre création, vous devez le signaler dans votre code source en utilisant un commentaire qui commence `ANTI-PLAGIAT`, et qui ensuite explique la provenence du code:

- **Un autre élève**
- **Un autre personne en dehors de l'ESEO**
- **Une site web**
- **Une intelligence artificielle (par exemple ChatGPT, Copilot, Gemini, ...)**

Un exemple d'un tel commentaire:

```
\*
 * ANTI-PLAGIAT: La méthode tourner() est basé sur le
 * travail de Martin Dupont.
 */
```

2.1 Notation

Le Projet est noté sur 20. Votre note sera composée de:

- Une note sur 14
  - La conformité de votre solution via-à-vis cet énoncé
  - La qualité de votre code
  - La couverture de votre code de production par les classes de test que vous avez implémenté
- Une note sur 6
  - Sur les extensions que vous implémentez une fois que vous avez terminé les instructions dans l'énoncé.
  - Cette note prend en compte la difficulté d'implémenter les extensions, la qualité de votre code et la démarche que vous avez fait (modélisation, implémentation, validation, documentation)

3 Travail demandé

Pour créer notre application, nous allons travailler d'une façon itérative. Dans un premier temps, nous allons implémenter la logique de base de notre application (la modèle), avant de commencer de créer l'interface graphique (la vue). Ensuite, nous allons revenir d'ajouter plus de fonctionnalité, modifier l'interface graphique et ajouter les interactions utilisateur (le contrôleur), jusqu'à nous arriverons à une application qui nous permet de jouer une version basique du jeu FallingBlox.

Diagrammes UML

Une partie du travail demandé est illustrée à l'aide de diagrammes UML:

- diagrammes de cas d'utilisation;
- diagrammes de classes;
- diagrammes d'activité; et
- diagrammes de séquences

À propos des diagrammes de classes

Il est important de noter que les diagrammes de classes présentent essentiellement les variables et méthodes dont la visibilité est **`public`** et **`protected`**. Certaines variables et méthodes dont la visibilité est **`private`** sont également représentées sur les diagrammes. Cependant, vous pourrez avoir besoin d'autres méthodes ou variables privées pour vos implémentations. N'hésitez pas à les définir, même si elles n'apparaissent pas dans les diagrammes.

Dans les diagrammes de classes, les nouvelles classes, attributs et méthodes sont écrits en **bleu foncé et en gras**. Les classes, attributs ou méthodes qui ont été déjà définis dans une section précédente sont écrits en noir. Les classes et interfaces déjà défini dans l'API de Java et que nous avons besoins d'utiliser (mais pas besoin d'implémenter) sont identifiées avec le texte en gris clair sur un fond gris foncé.

3.1 Préparation

Avant de commencer la programmation en Java, il est important de créer les répertoires dans lesquels vous rangerez votre code source et les classes Java qui résultent de vos compilations.

3.1.1 Le répertoire racine de votre travail

Le répertoire racine d'un système de fichiers est le répertoire se trouvant le plus haut dans l'arborescence des répertoires. Comme nous utilisons Eclipse pour écrire notre code, nous devons créer un *espace de travail* (*workspace* dans la terminologie Eclipse) dans lequel il pourra stocker tous les fichiers manipulés (les classes, les paquetages, les ressources, …). C'est dans cet espace de travail que nous créons les projets

Java. Pour chaque projet créé, eclipse créé un répertoire sur le système de fichiers. Ce répertoire sera le répertoire racine pour FallingBlox.

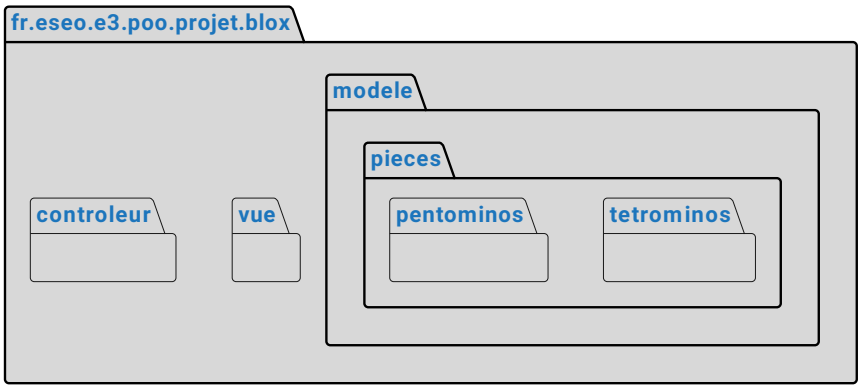


Figure 3: Les paquetages pour l'application FallingBlox

Le paquetage racine de notre application sera le paquetage **`fr.eseo.e3.poo.projet.blox`**. Il contiendra les paquetages **`modele`**, **`vue`** et **`controleur`** qui contiendront les classes relatives aux trois composantes de l'application.

Travail à faire

[titre=Travail à faire] Sous eclipse:

- créer un projet du type Java Project. S'assurer que l'option de créer les dossiers de source et de classe séparées est sélectionnée et que l'option de créer un fichier *module-info.java* n'est pas sélectionné.
- créer un deuxième dossier source qui s'appelle `test`
- dans les deux dossiers sources `src` et `test`, créer l'organisation hiérarchique des paquetages telle que représentée dans la figure 3. Par la suite, toute classe de production doit être définie dans le dossier `src` et toute classe de test **`XxxxTest`** doit être définie dans le dossier `test`.

3.1.2 Validation de votre travail

Pendant tout le déroulement de ce projet il vous sera demandé de valider votre travail avec l'Assignment Centre. Une application Java a été créée à cet effet: The Assignment Centre: Student Module. Elle utilise les principes de test unitaire et de test fonctionnel pour vérifier que votre application est conforme à ce qui est demandé dans l'énoncé. L'outil vérifie également:

- la qualité du code produit. Notamment en ce qui concerne le nommage, l'indentation, la complexité et les commentaires.
- la couverture de vos propres tests unitaires.

L'Assignment Centre ne se content pas de vous fournir une évaluation de votre travail, il crée un fichier que vous devez déposer sur le campus à la fin de chaque séance de travail et qui permettra d'évaluer votre progression. Une partie de votre note étant calculée à partir du travail que vous remettez. Il est primordial que vous compreniez bien le fonctionnement de l'application afin de soumettre une 'photographie' de votre travail.

Note 1: Lorsque votre travail est déposé sur le campus, il est analysé par des logiciels de détection de plagiat.

Note 2: En règle générale, si l'Assignment Centre juge que votre travail n'est pas conforme aux instructions, il fournit l'information utile pour vous aider à trouver les erreurs. En modifiant vos propres tests par exemple.

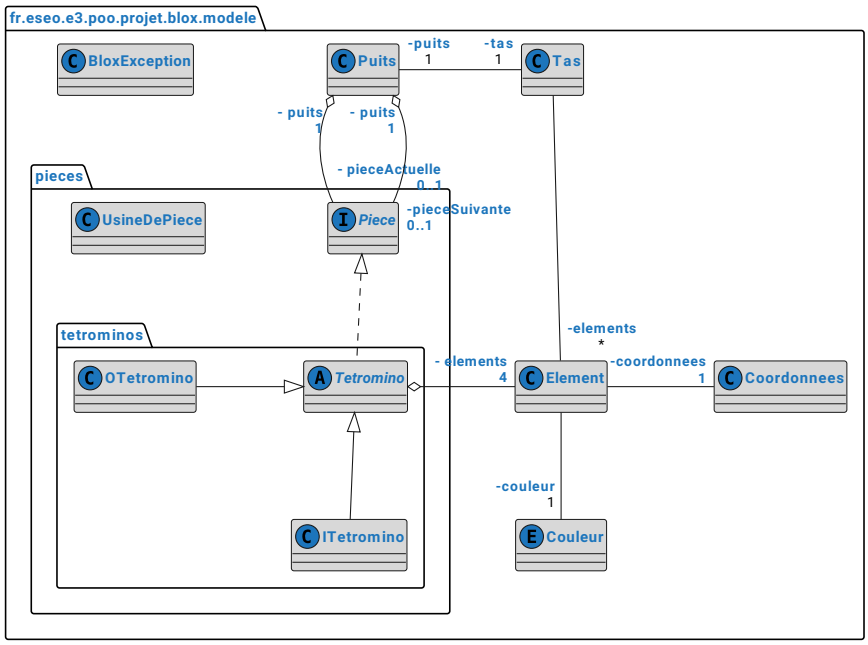


Figure 4: Diagramme de classe du paquetage modele à la fin de ce document (simplifié).

Note 3: L'Assignment Centre valide la conformité de vos classes compilées. Bien entendu, avant de le lancer, le code de votre projet doit être compilable et les erreurs doivent être corrigées.

Travail à faire

Valider votre travail avec l'Assignment Centre: 3.1

L'Assignment Centre vérifie que les dossiers représentant les paquetages se trouvent bien dans le dossier racine de votre projet.

3.2 Première implémentation de la logique du jeu)

La figure 4 décrit une version simplifiée du modèle qui sera utilisée pour Falling Blox. La classe **`Puits`** représente le plateau de jeu, qui peut contenir au plus un **`Tas`** et deux **`Pieces`** (la pièce actuelle et la pièce suivante). Pour l'instant ces **`Pieces`** peuvent être soit un **`OTetromino`** soit un **`ITetromino`**. Le **`Tas`** est l'ensemble des **`Elements`** qui sont empilés au fond du **`Puits`**. Une **`Piece`** soit un **`Tetromino`** soit un **`Pentomino`** est un ensemble d'**`Elements`** qui peuvent tomber dans le **`Puits`**. Chaque **`Element`** est définie par sa **`Couleur`** et ses **`Coordonnees`**. La classe **`UsineDePiece`** permet de construire une nouvelle **`Piece`**. Enfin la classe **`BloxException`** permettra d'identifier tout comportement exceptionnel lors du déplacement d'une **`Piece`** dans le **`Puits`**.

Dans un premier temps, nous ne nous intéresserons qu'à l'interface **`Piece`**, aux classes **`Coordonnees`**, **`Element`**, **`Tetromino`** (et ses sous-classes **`OTetromino`** et **`ITetromino`**), **`Puits`**,**`UsineDePuits`** et l'énumération **`Couleur`**. Les classes **`Tas`** et **`BloxException`** seront vues à partir de la section 3.4.

L'implémentation des autres sous-classes de **`Tetromino`** et de la classe **`Pentomino`** et de ses sous-classes, est laissé comme extensions.

3.2.1 L'énumeration Couleur

Dans le jeu de Falling Blox, une des caractéristiques qui identifie les différents **`Elements`** est sa couleur. Nous voulons restreindre le choix de la couleur. Pour le faire, nous utilisons une énumération.

Figure 5: L'énumération Couleur

Travail à faire

Définir l'énumération **`fr.eseo.e3.poo.projet.blox.modele.Couleur`** telle qu'elle est représentée dans le diagramme de figure 5.

Valider votre travail avec l'Assignment Centre: 3.2.1

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.2.2 La Classe Coordonnees

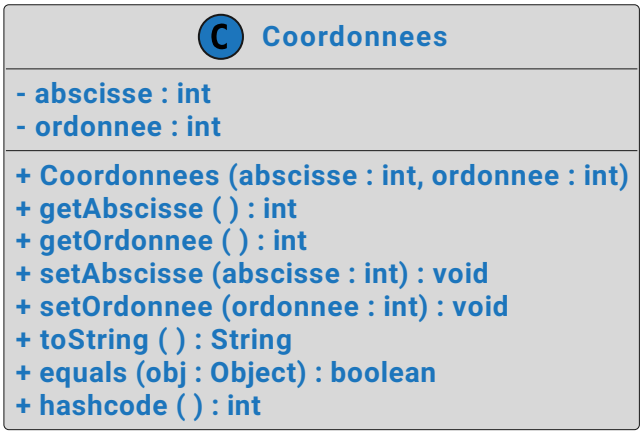


Figure 6: La classe Coordonnees

L'autre caractéristique qui identifie les différents **`Element`**s est ses coordonnées. Nous utilisons la classe **`Coordonnees`** comme représentée dans la figure 6. Pour nous aider dans l'écriture de nos classes, Eclipse peut générer automatiquement certains morceaux de nos classes (voir dans le menu Source).

Travail à faire

Définir la classe **`fr.eseo.e3.poo.projet.blox.modele.Coordonnees`** telle qu'elle est représentée dans le diagramme de figure 6.

- L'abscisse et l'ordonnée sont des entiers.
- Le constructeur crée une instance de **`Coordonnees`** dont l'abscisse et l'ordonnée sont données en argument.
- Les accesseurs et les mutateurs permettent de manipuler les variables d'instance.
- La méthode **`public String toString()`** est définie dans la classe **`Object`** de

retourner le nom de la classe suivi par une valeur hexadécimale (l'hashcode). Nous voulons redéfinir le fonctionnement de cette méthode pour la classe **Coordonnees** pour qu'elle retourne l'abscisse et l'ordonnée entre paranthèses et séparées par une virgule suivie par une espace.
Par exemple, la chaîne de caractères retournée par un appel à la méthode **toSring()** d'une instance de la classe **Coordonnees** créée avec **new Coordonnees(15, 12);** sera[a]:

(15,␣12)

- Dans FallingBlox, deux instances de la classe **Coordonnees** sont identiques si et seulement si l'abscisse, l'ordonnée sont égales. Nous devons redéfinir la méthode **public boolean equals(Object obj)** (de la classe **Object**) pour que cette affirmation est correcte[b].

[a] Dans tous les exemples de chaîne de caractère, les espaces sont identifiés par: ␣

[b] Eclipse comme écrit ci-dessus, peut générer cette méthode, en utilisant les variables d'instances pour l'abscisse et l'ordonnée. Quand Eclipse génère la redéfinition de la méthode **equals**, il redéfine la méthode **public int hashcode()** aussi. Ceci est très important, pour la bonne fonctionnalité de plusieurs classe dans Java. Dans les spécifications de Java, les variables d'instance qui sont utilisés pour calculer la résultat de **equals()** doivent être utilisées aussi pour le calcul de **hashcode()**. Voir https://www.baeldung.com/java-equals-hashcode-contracts

Valider votre travail avec l'Assignment Centre: 3.2.2

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.2.3 Vérifier la classe Coordonnees

Le fonctionnement des classes dans le modèle doit être validé par des tests JUnit.

Travail à faire

Créer, dans le paquetage **fr.eseo.e3.poo.projet.blox.modele** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **fr.eseo.e3.poo.projet.blox.modele.CoordonneesTest**

Implementer dans cette classe, les tests JUnit5 permettant de tester les fonctionnalités définies pour la classe **fr.eseo.e3.poo.projet.blox.modele.Coordonnees**.

- Exécuter votre classe **CoordonneesTest**
- Si necessaire, modifier votre
  - classe de production: **fr.eseo.e3.poo.projet.blox.modele.Coordonnees**
  - classe de test: **fr.eseo.e3.poo.projet.blox.modele.CoordonneesTest**

Valider votre travail avec l'Assignment Centre: 3.2.3

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.2.4 Créer un Element

Les **Element**s du jeu FallingBlox sont les plus petits objets disponibles, ils correspondent aux carrés dont sont constitués les **Pieces** et qui, par la suite, se retrouvent dans le **Tas** au fond du **Puits**. Chaque **Element** est caractérisé par ses **Coordonnees** et sa **Couleur**.

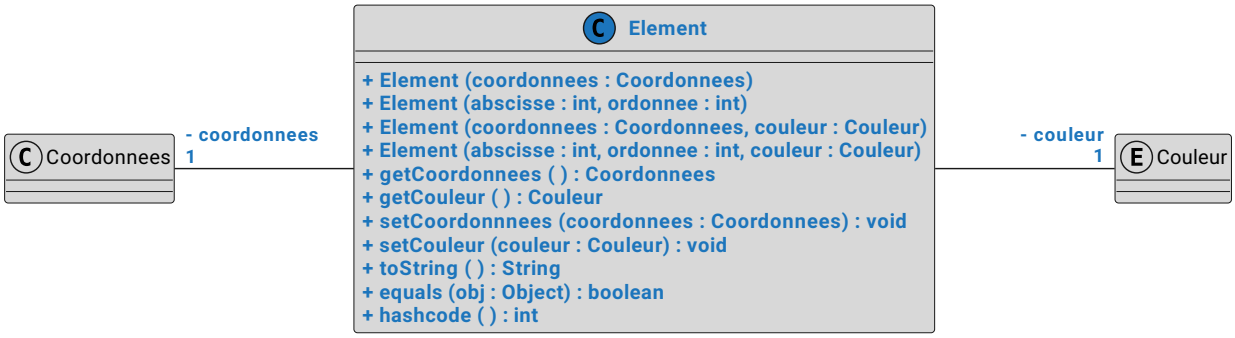


Figure 7: La classe Element

Travail à faire

Créer la classe **`Element`** du paquetage **`fr.eseo.e3.poo.projet.blox.modele`** telle qu'elle est représentée par le diagramme de classe de la figure 7.

- Les constructeurs se chargent de créer une instance de la classe avec une valeur pour les variables d'instance **`coordonnees`** et **`couleur`**, pour les deux constructeurs où il n'y a pas un paramètre pour la couleur, la première couleur de l'énumération **`Couleur`** doit être utilisée[a].
- Les accesseurs et les mutateurs permettent de manipuler les variables d'instance.
- comme pour la classe **`Coordonnees`** nous voulons redéfinir la fonctionnalité de la méthode **`toString()`**. La chaîne de caractères générée doit afficher sur une ligne les coordonnées de l'**`Element`** suivi par le nom du **`Couleur`**, séparé par une espace, un tiret haut et une autre espace.
  Par exemple, la chaîne de caractères retournée par un appel à la méthode **`toString()`** d'une instance de la classe **`Element`** créée avec **`new Element(12,7, Couleur.VIOLET);`** sera:
  `(12, 7) - VIOLET`

  Redéfinir les méthodes **`equals()`** and **`hashcode()`** pour qu'elles utilisent les variables d'instance couleur et coordonnees pour faire leur calcul.

[a] La méthode de classe **`values()`** de la classe **`enum`** sera utile.

Valider votre travail avec l'Assignment Centre: 3.2.4

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.2.5 Vérifier la classe Element

Le fonctionnement des classes dans le modèle doit être validé par des tests JUnit.

Travail à faire

Créer, dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`fr.eseo.e3.poo.projet.blox.modele.ElementTest`**

Implémenter dans cette classe, les tests JUnit5 permettant de tester les fonctionnalités définies pour la classe **`fr.eseo.e3.poo.projet.blox.modele.Element`**.

- Exécuter votre classe **`ElementTest`**
- Si necessaire, modifier votre
  - classe de production:
    `fr.eseo.e3.poo.projet.blox.modele.Element`
  - classe de test:
    `fr.eseo.e3.poo.projet.blox.modele.ElementTest`

Valider votre travail avec l'Assignment Centre: 3.2.5

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.2.6 Créer la première Tetromino: L'OTetromino

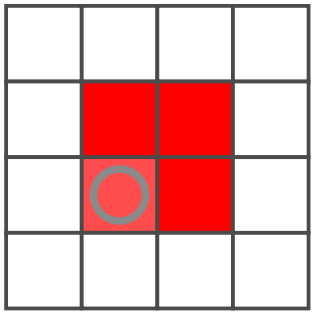


Figure 8: Une OTetromino

Les **Pieces** (voir figure 1 sont généralement les **Tetromino** qui sont constituées de quatre **Elements**, disposés selon différentes configurations[1]. Chaque **Tetromino** est caractérisée par un nom pour l'identifier, ce nom est **?Tetromino** où le point d'interrogation est remplacé par la lettre associée à la pièce ressemblante.

Un des **Elements** est désigné comme l'élément de référence d'une **Piece**. Les coordonnées de cet élément de référence sont utilisés pour définir la position d'une **Piece** dans le **Puits**. Dans les figures de ce document, l'élément de référence est d'une couleur légèrement plus vive que les autres avec un cercle gris. Les coordonnées des **Elements** de la **Piece** sont définies par leurs positions relatives par rapport à l'élément de référence.

Dans un premier temps, nous ne nous intéressons qu'aux **OTetromino** montrée dans la figure 8. Supposant que l'origine $(0, 0)$ du repère soit le carré situant dans le coin supérieur gauche[2], les coordonnées de la **OTetromino** de la figure 8 sont $(1, 2)$, ce sont les coordonnées de l'élément de référence de la OTetromino, les autres Eléments se trouvent aux coordonnées: $(1, 1)$, $(2, 2)$ et $(2, 1)$.

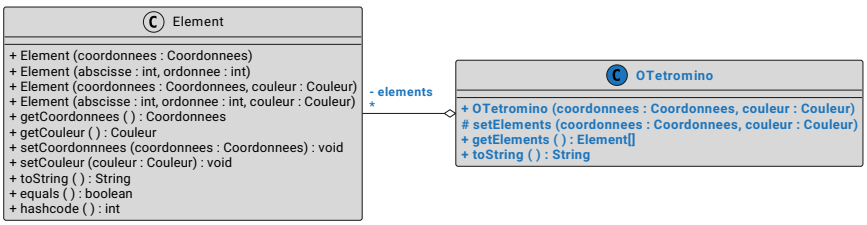


Figure 9: La classe OTetromino

Travail à faire

Créer la classe **OTetromino** dans le paquetage **fr.eseo.e3.poo.projet.modele.pieces.tetrominos** telle qu'elle est réprésentée par le diagramme de classe de la figure 9.

- La rélation entre **OTetromino** et **Element** doit être représenté par un tableau de quatre **Elements**.

[1] Des variantes du jeu de Tetris utilisent des pièces constituées de plus de quatre éléments. Ce sont assez souvent des pièces bonus qui apparaissent à la fin de certains niveaux.

[2] Dans la majorité des systèmes graphiques informatique, l'origine du repère se trouve dans le coin supérieur gauche, l'axe des x est un axe horizontal dirigé vers la droite et l'axe des y est un axe vertical dirigé vers le bas. Plus d'information dans la section 3.3.1.

- Le constructeur prend en argument les coordonnées de l'élement de référence et la couleur de la pièce. Le constructeur initialise le tableau d'**`Element`** (qui est immuable), qui va être utilisé pour stocker les **`Element`**s de la **`OTetromino`**, et ensuite fait appel à la méthode **`setElements(…)`** pour remplir ce tableau.
- La méthode **`protected void setElements(…)`** doit remplir le tableau des **`Element`**s avec quatre nouvelles instances de la classe **`Element`** en utilisant la couleur et les coordonnees de référence passée en argument. La première **`Element`** dans le tableau doit être l'élément de référence, les autres **`Element`**s peuvent être ajoutées dans n'importe quel ordre.
- La méthode **`getElements()`** retourne le tableau des **`Element`**s.
- La redéfinition de la méthode **`toString()`** permet de générer une chaine de caractères sur plusieurs lignes. Sur la première ligne, le nom de la **`Tetromino`** suivi par une espace et un deux points, ensuite pour chaque **`Element`** (dans l'ordre stocké dans le tableau), une ligne qui commence par une tabulation suivie par le résultat d'un appel de sa méthode **`toString()`**.
  Par exemple, la chaîne de caractères retournée par un appel à la méthode **`toString()`** d'une instance de la classe **`OTetromino`** créée avec **`new OTetromino(new Coordonnees(6,5), Couleur.CYAN);`** peuvent être:[a]:

```
OTetromino :
    (6, 5) - CYAN
    (7, 5) - CYAN
    (6, 4) - CYAN
    (7, 4) - CYAN
```

[a] Les tabulations sont identifiées par ⟶.

Valider votre travail avec l'Assignment Centre: 3.2.6

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.2.7 Vérifier la classe OTetromino

Le fonctionnement des classes dans le modèle doit être validé par des tests JUnit.

Travail à faire

Créer, dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos.OTetrominoTest`**

Implémenter dans cette classe, les tests JUnit5 permettant de tester les fonctionnalités définies pour la classe **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos.OTetromino`**.

- Exécuter votre classe **`OTetrominoTest`**
- Si necessaire, modifer votre
  - classe de production **`fr.eseo.e3.poo.projet.blox.modele.pieces.OTetromino`**
  - class de test **`fr.eseo.e3.poo.projet.blox.modele.pieces.OTetrominoTest`**

Valider votre travail avec l'Assignment Centre: 3.2.7

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.2.8 Refactorisation du code pour avoir d'autres type de Piece

La refactorisation[3] du code consiste à retravailler le code source d'un programme, pour améliorer sa qualité, sa maintenabilité ou sa lisibilité. Les approches de développement Agile font une utilisation intensive de la refactorisation de code dans le but de promouvoir des niveaux de qualité de code croissant.

Le jeu FallingBlox serait plus simple si toute **`Piece`** était du type **`OTetromino`**. Pour rendre le jeu un peu plus intéressant, nous allons maintenant implémenter un nouveau type de Piece. Dans un premier temps, nous n'ajouterons qu'un seul nouveau type de **`Piece`**, les **`ITetromino`** (la plus simple après **`OTetromino`**).

Toutes les **`Piece`**s ont en commun un certain nombre de fonctionnalités, néanmoins elles diffèrent par leurs formes, par certaines de leurs fonctionnalités et dans les extensions par le nombre de leur **`Element`**s. Nous proposons de créer deux niveaux d'abstraction:

- Créer une interface **`Piece`**, qui fourni le contrat qui va garantir le comportement de toutes les différentes **`Piece`**s
- Créer une classe abstraite Tetromino pour factoriser les aspects structurels et fonctionnelles communes à toutes les **`Tetromino`**s

Ceci nous permettra, lorsqu'on souhaitera définir un nouveau type de Tetromino, de créer une sous-classe de la classe **`Tetromino`** pour préciser les parties structurelles et fonctionnelles de ces nouvelles **`Tetrominos`**s, ou (dans les extenstions) nous permettra de définir la classe abstraite **`Pentomino`** et ses sous-classes pour préciser les parties structurelles et fonctionnelles de ces nouvelles **`Pentomino`**s. La figure 10 montre les modifications qui doivent être apportées à l'arbre d'héritage.

[3] nos amis québecois parlent aussi parfois de réusinage de code et les anglophones de code refactoring

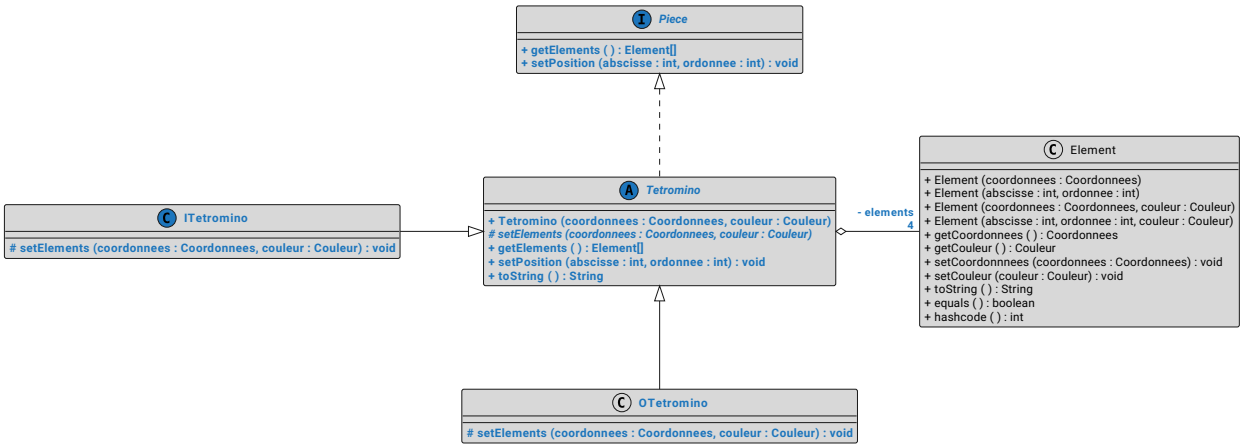


Figure 10: Les différentes Pieces

Travail à faire

Créer l'interface **`Piece`** dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele.pieces`** telle qu'elle est réprésentée par le diagramme de classe de la figure 10.

- Ajouter la signature des méthodes **`getElements()`** et **`setPosition()`**

Créer la classe *abstraite* **`Tetromino`** qui doit implémenter l'interface **`Piece`** dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos`** telle qu'elle est réprésentée par le diagramme de classe de la figure 10.

- Refactoriser (déplacer / renommer / copier / réécrire où nécessaire) le constructeur et la méthode **`toString()`** de la classe **`OTetromino`** pour les intégrer à cette nouvelle classe.
- Ajouter la signature de la méthode *abstraite* **`setElements()`** à la classe **`Tetromino`**
- Déplacer la méthode **`getElements()`** et la variable d'instance associée de la classe **`OTetromino`** vers la classe **`Tetromino`**.
- Créer la méthode **`setPosition(int abscisse, int ordonnee)`** qui doit modifier la position de la **`Tetromino`**, en assurant que l'**`Element`** de référence se trouve aux coordonnées données en paramètre.

Réécrire la classe **`OTetromino`** de telle sorte à ce que:

- la classe **`OTetromino`** hérite de la classe **`Tetromino`**
- le constructeur de la classe **`OTetromino`** fasse référence au constructeur de la surclasse.

Créer la classe **`ITetromino`** en vous assurant que:

- elle hérite de la classe **`Tetromino`**
- elle propose le constructeur faisant référence au constructeur de la classe **`Tetromino`**
- elle implémente la méthode abstraite **`setElement()`** en suivant les consignes vu dans la section 3.2.6. Voir aussi la figure 1 pour identifier l'**`Element`** de référence.

Valider votre travail avec l'Assignment Centre: 3.2.8

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.2.9 Vérifier les classes ITetromino et OTetromino

Le fonctionnement des classes dans le modèle doit être validé par des tests JUnit.

Travail à faire

Créer, dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`ITetrominoTest`**

Implémenter dans cette classe, les tests JUnit5 permettant de tester les fonctionnalités définies pour la classe **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos.ITetromino`**.

- Exécuter vos classes **`ITetrominoTest`** et **`OTetrominoTest`**
- Modifer vos classes de production:
  - **`ITetromino`**
  - **`OTetromino`**

  et de test:
  - **`ITetrominoTest`**
  - **`OTetrominoTest`**

  si necessaire.

Valider votre travail avec l'Assignment Centre: 3.2.9

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.
Attention, parce que nous avons modifié la classe **`OPiece`**, c'est possible que l'Assignment Centre signale des problèmes pour la section 3.2.6. Si c'est le cas, la refactorisation a modifié la fonctionnalité de la classe **`OPiece`**, nous devons corriger ces problèmes.

3.2.10 Le Puits

La classe **`Puits`** modélise le plateau de notre jeu (celui sur lequel les différentes **`Pieces`** vont se déplacer). Les différentes **`Pieces`** qui ont été créées doivent pouvoir être ajoutées sur le plateau de jeu. Dans une section à venir (section 3.4 nous ajouterons les fonctionnalités qui permettront aux **`Pieces`** de se déplacer dans le **`Puits`**. Le **`Puits`** est défini par sa largeur et sa profondeur (ces informations seront données en nombre de carreaux). Sa largeur et sa hauteur doivent respecter les limites suivantes:

$$\text{largeur:} \in \mathbb{N}, \text{largeur: } \in [5, 15] \text{ profondeur:} \in \mathbb{N}, \text{profondeur: } \in [15, 25]$$

Les constructeurs et les mutateurs doivent lancer une exception de type **`IllegalArgumentException`** si ses limites ne sont pas respectés. Le **`Puits`** doit aussi connaitre la **`pieceActuelle`** qui est en train de chuter, ainsi que celle qui devra être affichée lorsque la pieceActuelle se sera immobilisée au fond du **`Puits`**, la **`pieceSuivante`**.

Travail à faire

Créer la classe **`Puits`** dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele`** telle qu'elle est représentée dans la figure 11.

- Le constructeur *par défaut* doit créer une instance de **`Puits`** dont les dimensions sont la largeur et la profondeur par défaut telles que spécifieés par les constantes de classe **`LARGEUR_PAR_DEFAUT`** et **`PROFONDEUR_PAR_DEFAUT`**.
  Le deuxième constructeur devra, quant à lui, créer un **`Puits`** de largeur et profondeur données.
- Les accesseurs **`getPieceActuelle`** et **`getPieceSuivante`** permettent de recuperer les **`Pieces`** associées avec le **`Puits`**.
- Le mutateur **`setPieceSuivante`** doit en premier vérifier s'il y a déjà défini une pieceSuivante. Si c'est le cas, cette méthode doit définir la pieceActuelle comme

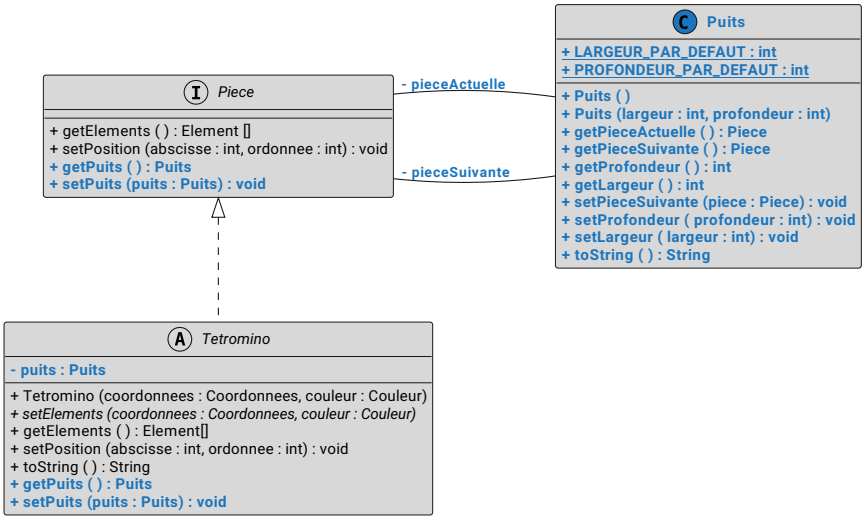


Figure 11: La classe Puits

celle déjà stocker dans pieceSuivante. La position de la pieceActuelle doit être modifiée (en utilisant la méthode **`setPosition()`** pour qu'elle se trouve aux coordonnées $(\frac{largeurdepuits}{2}, -4)$. Dans les deux cas, la pieceSuivante de cette instance de **`Puits`** et remplacée avec celle donnée en paramètre.

- Les accesseurs et mutateurs pour la largeur et la profondeur permettent de manipuler les variables d'instance.
- la redéfinition de la méthode **`toString()`** retourne une chaîne de caractères sur plusieurs lignes. Sur la première ligne, il y a Puits suivi par sa dimension, sur les prochaines lignes il y a les informations sur la pieceActuelle suivi par la pieceSuivante. Si jamais la pieceActuelle et/ou la pieceSuivante n'est pas encore défini, l'information sur la piece et remplacé par la chaîne: **`<aucune>`**. Le formatage de la chaîne de caractères doit respecter l'exemple suivant (pour un Puits de largeur 10 et profondeur 15, avec que la pieceSuivante définie):

```
Puits : Dimension 10 x 15
Piece Actuelle : <aucune>
Piece Suivante : ITetromino :
      (7, 8) - ROUGE
      (7, 9) - ROUGE
      (7, 6) - ROUGE
      (7, 7) - ROUGE
```

Modifier l'interface **`Piece`** dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele.pieces`** et la classe abstraite **`Tetromino`** dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos`** telle qu'elles sont représentées dans la figure 11.

- Ajouter la variable d'instance et son mutateur et son accesseur.

Valider votre travail avec l'Assignment Centre: 3.2.10

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.2.11 Vérifier la classe Puits

Le fonctionnement des classes dans le modèle doit être validé par des tests JUnit.

Important!

Lorsque l'assignment centre valide votre travail, vous rencontrez une erreur avec la méthode **`setPieceSuivante`**, et que la message est similaire à "Après un deuxième appel à setPieceSuivante, la nouvelle pieceActuelle n'est pas correcte, coordonnees pas correctes". , il est fort probable que vous deviez modifier votre méthode **`setElements()`** dans *X*Tetromino. **Assurez-vous de bien utiliser le constructeur Element(int, int, Couleur) pour la création de chaque élèment pour la Piece concernée.**

Travail à faire

Créer, dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`fr.eseo.e3.poo.projet.blox.modele.PuitsTest`**

Implémenter dans cette classe, les tests JUnit5 permettant de tester les fonctionnalités définies pour la classe **`fr.eseo.e3.poo.projet.blox.modele.Puits`**. Modifier, dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos`** qui se trouve dans le dossier source test, les cas de test JUnit 5 suivants:

- **`OTetrominoTest`**
- **`ITetrominoTest`**

Implémenter dans cette classe, les tests JUnit5 permettant de tester les nouvelles fonctionnalités définies pour les classes **`OTetromino`** et **`ITetromino`**.

- Exécuter vos classes **`PuitsTest`**, **`OTetrominoTest`** et **`ITetrominoTest`**
- Modifer vos classes de productions:
  - **`Puits`**
  - **`OTetromino`**
  - **`ITetromino`**

  et de tests:
  - **`PuitsTest`**
  - **`OTetrominoTest`**
  - **`ITetrominoTest`**

  si necessaire.

Valider votre travail avec l'Assignment Centre: 3.2.11

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.2.12 L'UsineDePiece

Au cours du déroulement du jeu, à chaque fois qu'une **`Piece`** atteint le fond du **`Puits`**, une nouvelle **`Piece`** devra commencer à tomber dans le **`Puits`**. Pour l'instant ces **`Piece`**s sont choisis parmi les **`Tetrominos`** déjà implémentés. Au fur et au mesure des extensions, cette classe devrait être modifiée pour permettre de choisir parmi toutes les **`Tetromino`** (et aussi parmi les **`Pentominos`**...). La création d'une **`Piece`** sera gérée par la classe **`UsineDePiece`** représentée dans la figure 12. L'idée de cette classe est de fournir des méthodes statiques qui permettent de créer une **`Tetromino`** suivant un des trois façons:

- Complètement aléatoire
  Choisir un type de tetromino aléatoirement, en utilisant une couleur aléatoire

- Type aléatoire
  Choisir un type de tetromino aléatoirement, en utilisant la même couleur (comme défini dans la figure 1) pour la même type de tetromino

- Cyclique

Choisir dans l'ordre les tetrominos (OTetromino, puis ITetromino, ..., OTetromino, ...) comme décrit dans l'étiquette de la figure 1 en respectant les couleurs utilisées dans la figure 1[4].

L'**`UsineDePiece`** utilisera une instance de la classe **`java.util.Random`** et sa méthode **`nextInt(int)`** pour choisir une valeur entière aléatoire.

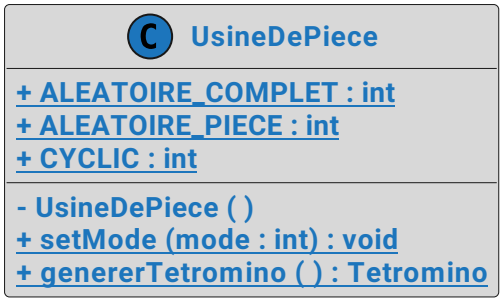


Figure 12: La classe UsineDePiece

Travail à faire

Créer la classe **`UsineDePiece`** dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele.pieces`** telle du'elle est représentée dans la figure 12.

- Le constructeur *privé par défaut* doit être implémenté avec aucune fonctionnalité. Il servira d'interdire la création d'une instance de cette classe, car toutes les méthodes sont les méthodes de classe (*statique*).
- Les constantes de classe **`CYCLIC`**, **`ALEATOIRE_COMPLET`** et **`ALEATOIRE_PIECE`** sont utilisés pour stocker une valeur entière pour définir les trois modes de fonctionnement de l'**`UsineDePiece`**
- La méthode de classe **`setMode(int)`** permettra de choir la mode de fonctionnement de l'**`UsineDePiece`**. Si la mode **`CYCLIC`** est choisi, vous devez assurer que la prochaine tetromino généré par l'usine doit être une instance d'**`OTetromino`**.
- La méthode de classe **`genererTetromino()`** retourne une **`Tetromino`** avec coordonnées $(2, 3)$[a]. Le type et la couleur de la **`Tetromino`** générés dépendent la mode de fonctionnement choisi.
- par défaut la mode de fonctionnement doit être **`ALEATOIRE_PIECE`**.

[a] Pourquoi $(2, 3)$? Pour comprendre, finir jusqu'à la section 3.8.5...

Valider votre travail avec l'Assignment Centre: 3.2.12

L'Assignment Centre vérifie que la structure de votre code est correcte.

[4] Chaque fois que nous implémentons une nouvelle piece, nous avons besoin de l'ajouter en respectant toujours cet ordre (en commençant en haut à gauche et en lisant comme les lignes d'un livre: O, I, T, L, J, ...

3.2.13 Vérifier la classe UsineDePiece

Le fonctionnement des classes dans le modèle doit être validé par des tests JUnit.

Travail à faire

Créer, dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`fr.eseo.e3.poo.projet.blox.modele.UsineDePieceTest`**

Implémenter dans cette classe, les tests JUnit5 permettant de tester les fonctionnalités définies pour la classe **`fr.eseo.e3.poo.projet.blox.modele.UsineDePiece`**.

- Exécuter votre classe **`UsineDePieceTest`**
- Modifer votre classe de production:
  - **`fr.eseo.e3.poo.projet.blox.modele.UsineDePiece`**

  et de test:
  - **`fr.eseo.e3.poo.projet.blox.modele.UsineDePieceTest`**

  si necessaire.

Valider votre travail avec l'Assignment Centre: 3.2.13

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.3 Création de notre première interface graphique

Les éléments du modèle que nous avons créés pour le moment sont suffisants pour qu'il soit possible d'en proposer une représentation graphique dans la vue. À ce stade, le modèle n'est, en aucun cas, terminé et il sera nécessaire, d'y ajouter plus d'éléments et de fonctionnalités. Cependant, afin de nous donner un premier aperçu du modèle de données, nous proposons de définir les premiers éléments de la vue.

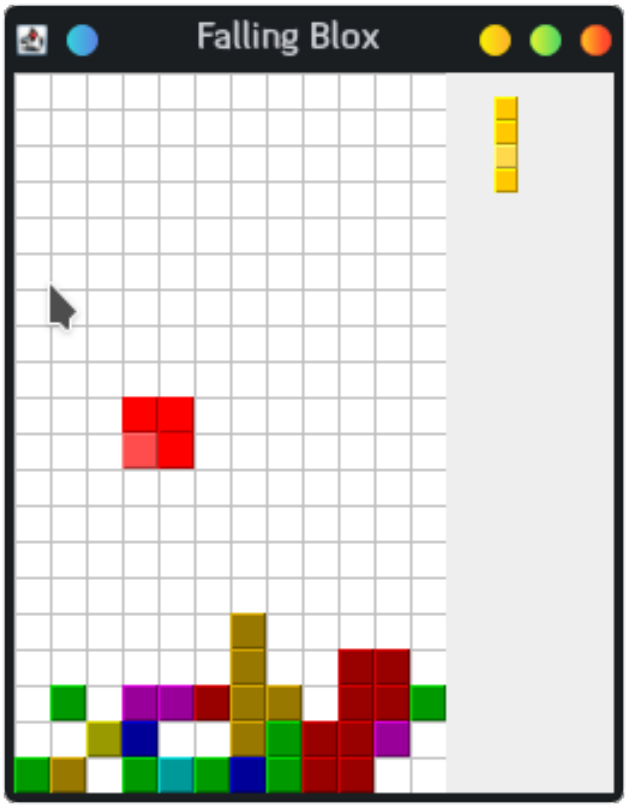


Figure 13: L'application finale

Les classes de la vue comprendront le code qui permettra de représenter graphiquement le **`Puits`** et la

Piece actuelle. Un exemple de la fenêtre de l'application à la fin de ce document est illustré dans la figure 13

3.3.1 La classe VuePuits

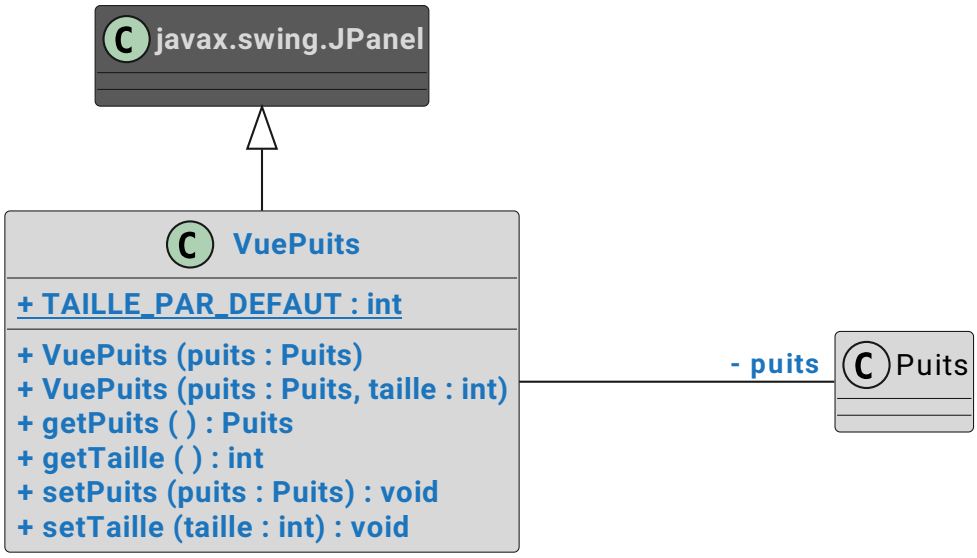


Figure 14: La classe VuePuits

Le **VuePuits** est le composant qui sera utilisé pour fournir une représentation graphique du **Puits** à l'utilisateur. C'est également à travers la classe **VuePuits** que l'utilisateur pourra interagir (en utilisant sa souris) avec la **Piece** qui tombe. Nous définirons la classe **VuePuits** en tant que sous-classe de **javax.swing.JPanel**. Dans un premier temps, nous concentrons de créer une interface avec la bonne taille.

Travail à faire

Écrire la classe **fr.eseo.e3.poo.projet.blox.vue.VuePuits** sous-classe de **javax.swing.Jpanel** telle qu'elle est représentée dans la figure 14

- Le constructeur avec un seul paramètre sera chargé de créer l'association entre le **Puits** et le **VuePuits** et utilisera la constante de classe **TAILLE_PAR_DEFAUT** pour définir, en pixels, la taille à laquelle sera représentée un **Element** dans ce **VuePuits**.
- Le constructeur avec deux paramètres fera la même chose que le premier sauf s'il utilise la taille donnée comme argument pour définir, en pixels, la taille à laquelle sera représentée un **Element**.
- Les accesseurs permettent de récupérer les valeurs stockées dans les variables d'instance.

Valider votre travail avec l'Assignment Centre: 3.3.1

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.3.2 Vérifier la classe VuePuits

Travail à faire

Créer, dans le paquetage **fr.eseo.e3.poo.projet.blox.vue** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **fr.eseo.e3.poo.projet.blox.vue.VuePuitsTest**

Implémenter dans cette classe, les tests JUnit5 permettant de tester les fonctionnalités définies pour la classe **fr.eseo.e3.poo.projet.blox.modele.vue.VuePuits**.

- Exécuter votre classe **VuePuitsTest**
- Modifer votre classe de production (**fr.eseo.e3.poo.projet.blox.vue.VuePuits**) et de test (**fr.eseo.e3.poo.projet.blox.vue.VuePuitsTest**) si necessaire.

Valider votre travail avec l'Assignment Centre: 3.3.2

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.3.3 Dessiner sur la VuePuits

Le rôle de la classe **VuePuits** est d'afficher la représentation actuelle du **Puits** - la pièce actuelle et les élements dans le tas en bas du puits. Afin d'aider le joueur, le **VuePuits** affichera aussi, une grille montrant les positions possibles des éléments sur le **Puits**. Au début, nous n'allons afficher que cette grille.

La méthode **paintComponent(Graphics g)**, définie dans la classe **javax.swing.JPanel** est une des trois méthodes que swing utilise pour dessiner les composants sur l'écran. Pour Falling Blox, c'est la seule qui nous intéresse. Elle est responsable d'afficher le contenu du JPanel. Le paramètre de la méthode **paintComponent** est un *contexte graphique* de type **java.awt.Graphics**. Ce paramètre est capturé par Java et transmis automatiquement à la méthode **paintComponent()**. Depuis les premières versions de Java, des améliorations on tété apportées aux primitives graphiques du langage. La plupart de ces améliorations ont été inplémentées dans une sous-classe de **Graphics**: La classe **java.awt.Graphics2D**. Cette classe propose des méthodes permettant un graphisme étendu, accéléré et de meilleure qualité. C'est la classe que nous utiliserons pour dessiner le VuePuits. Il sera nécessaire de transformer l'objet **java.awt.Graphics** transmis en paramètre de la méthode **paintComponent** en un objet équivalent de type **java.awt.Graphics2D** en créant une copie de l'instance et en la transtypant vers la sous-classe. Une fois que nous avons fini de dessiner, nous devons supprimer la copie. Ceci se fera de la façon suivante:

```
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    /* appel vers super pour remplir le fond du JPanel */

    /*Le paramètre g est copie en utilisant la méthode copie()
     * puis converti en instance de Graphics2D grâce à
     * un transtypage (cast) explicite.
     */
    Graphics2D g2D = (Graphics2D)g.create();
    /* Nous utiliserons l'instance de Graphics2D*/

    /*Puis nous liberons la memoire*/
    g2D.dispose();
}
```

Travail à faire

Modifier la classe **fr.eseo.e3.poo.projet.blox.vue.VuePuits** telle qu'elle est représentée dans la figure 15:

- Modifier les constructeurs et les mutateurs pour qu'ils définissent la taille de préférence pour le composant, en utilisant un appel de la méthode **setPreferredSize(Dimension dimension)** de la classe **JPanel**. Faire en

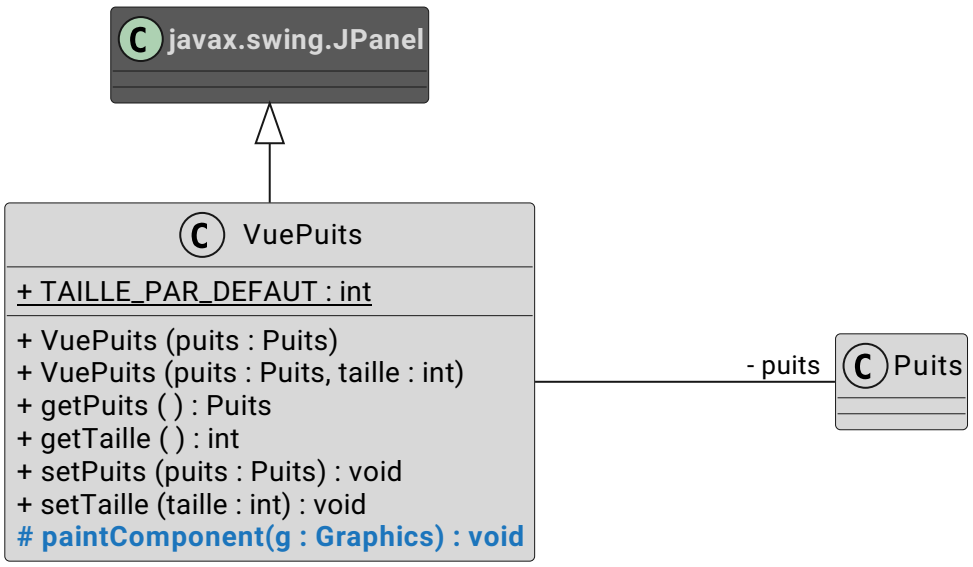


Figure 15: La classe VuePuits

sorte que le fond du composant soit blanc (**`java.awt.Color.WHITE`**).

- Redéfinir la méthode **`protected paintComponent(Graphics g)`** de façon qu'elle remplisse le fond du composant en blanc (grâce à un appel vers **`super.paintComponent()`**, ensuite qu'elle affiche une grille de couleur gris clair (**`java.awt.Color.LIGHT_GRAY`**). La taille des carreaux de la grille dépendra de la taille des éléments telle qu'elle est définie dans les constructeurs.

Valider votre travail avec l'Assignment Centre: 3.3.3

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.3.4 Vérifier l'interface homme machine

Il est difficile de tester les interfaces graphiques automatiquement avec JUnit, sans utiliser les bibliothèques spécialisées[5]. L'utilisation de telles bibliothèques n'est pas abordée en E3 et donc pour les tests de l'interface graphique, nous utilisons les applications Java, que nous exécuterons manuellement.

Les interfaces homme machine et la bibliothèque swing Pour que les panneaux puissent être affichés sur l'écran, ils doivent être ajoutés dans une fenêtre. La bibliothèque swing permet de créer trois types de fenêtres:

- les **`JFrame`**s
  - utilisés pour les fenêtres principales
- les **`JDialog`**s
  - utilisés pour les boîtes de dialogue
- les **`JWindow`**s
  - utilisés pour les fenêtres sans décoration du système d'exploitation (par exemple les *splash-screens*

Par la suite, nous utilisons les **`JFrame`**s pour nos tests.

[5] L'Assignment Centre utilise la bibliothèque Jemmy.

Presque tout le code qui crée ou interagit avec les composants swing doit s'exécuter sur l'*Event Dispatch Thread (EDT)* - thread de répartition des événements. Pour cette raison, les méthodes qui testent l'interface graphique doivent, elle aussi, être exécutées sur ce thread.

Il est possible de faire cela en utilisant la méthode de classe **`invokeLater()`** de la classe **`SwingUtilities`** qui prend en paramètre une cible d'une **`Thread`**, comme illustré par le code suivant[6]:

```
/* ... */
import javax.swing.SwingUtilities;

public class IhmTest {
    public IhmTest() {
        testPanneau();
    }

    public void testPanneau () {
        /* Le code de test */
    }

    public static void main (String [] args) {
        SwingUtilities.invokeLater(new Runnable () {
            @Override
            public void run() {
                new IhmTest ();
            }
        });
    }
}
```

Travail à faire

Avant de créer le test manuel, si besoin modifier la classe:
`fr.eseo.e3.poo.projet.blox.vue.VuePuitsTest` pour vérifier la nouvelle fonctionnalité (vérifier la taille de préférence...)
Créer, dans le paquetage **`fr.eseo.e3.poo.projet.blox.vue`** qui se trouve dans le dossier source test, la classe suivante:

- **`fr.eseo.e3.poo.projet.blox.vue.VuePuitsAffichageTest`**
- En se basant du code ci-dessous, écrire les méthodes de test:
  - **`private testConstructeurPuits()`** cette méthode doit créer un **`JFrame`** avec le titre Puits, ajouter physiquement une instance de **`VuePuits`** créée en utilisant un appel de **`new VuePuits(Puits)`**, modifier la taille de la fenêtre pour qu'elle prenne en compte la taille de préférence du **`VuePuits`**, modifier la position de la fenêtre pour qu'elle soit au centre de l'écran. Avant d'afficher la fenêtre, cette méthode doit s'assurer que l'application arrêtera quand l'utilisateur ferme la fenêtre.
  - **`private testConstructeurPuitsTaille()`** cette méthode doit créer un **`JFrame`** avec le titre Puits et taille, ajouter physiquement une instance de **`VuePuits`** créée en utilisant un appel de **`new VuePuits(Puits, int)`**, modifier la taille de la fenêtre pour qu'elle prenne en compte la taille de préférence du **`VuePuits`**, modifier la position de la fenêtre pour qu'elle est au centre de l'écran. Avant d'afficher la fenêtre, cette méthode doit s'assurer que l'application s'arrêtera quand l'utilisateur fermera la fenêtre.
- Lancer l'application de test et vérifier manuellement que deux fenêtres s'affichent sur l'écran avec les bons titres et les bonnes dimensions et que dans chaque fenêtre, une grille de couleur gris clair s'affiche sur un fond blanc.

[6] voir http://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html et http://docs.oracle.com/javase/8/docs/api/-javax/swing/SwingUtilities.html#invokeLater-java.lang.Runnable.

- Modifier votre classe de production (**`fr.eseo.e3.poo.projet.blox.vue.VuePuits`**) et de test (**`fr.eseo.e3.poo.projet.blox.vue.VuePuitsAffichageTest`**) si necessaire.

Valider votre travail avec l'Assignment Centre: 3.3.4

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

Attention, ne touchez pas le clavier ni la souris pendant l'exécution de tests par l'Assignment Centre. Lors des tests de l'interface graphique, l'Assignment Centre affiche les fenêtres sur l'écran et peut interagir avec. Si vous manipulez le clavier ou la souris pendant ces tests, l'Assignment Centre peut enregistrer des erreurs.

3.3.5 Modifier l'énumération Couleur

Comme nous avons vu dans la section 3.3.3 les couleurs utilisées pour l'affichage sont représentées par les instances de la classe **`java.awt.Color`**. Nous allons modifier notre énumération **`Couleur`** pour que chaque option sera liée avec une couleur de type **`java.awt.Color`**.

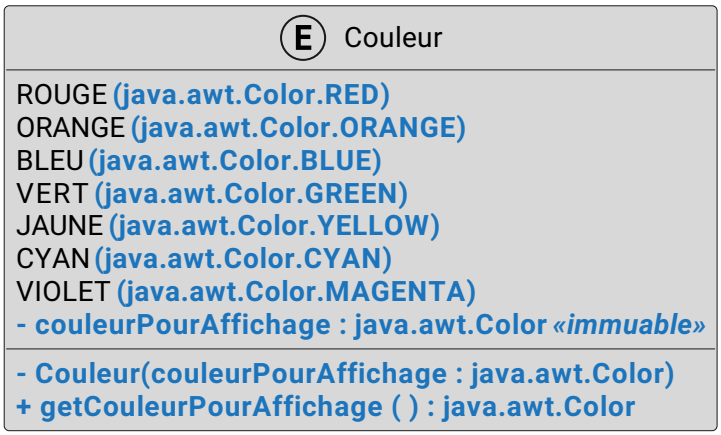


Figure 16: L'énumération Couleur après refactorisation

En java, nous pouvons ajouter d'autres informations à chaque constante définie dans un **`enum`**, en ajoutant entre parenthèses le(s) argument(s). Nous avons besoin de créer de(s) variable(s) d'instance pour stocker ces valeurs, de créer un constructeur privé pour initialiser ces variables et des accesseurs pour accéder à ces valeurs.

Travail à faire

Modifier l'énumération **`fr.eseo.e3.poo.projet.blox.modele.Couleur`** telle qu'elle est représentée dans la figure 16.

- Ajouter les valeurs de type **`java.awt.Color`** pour chaque constant.
- Créer une variable d'instance capable de stocker l'instance de **`java.awt.Color`**.
- Créer le constructeur privé qui permet d'enregistrer l'instance de **`java.awt.Color`** dans la variable d'instance.
- Créer la méthode **`getCouleurPourAffichage()`** qui permet d'accéder à la valeur stockée dans cette variable.

Valider votre travail avec l'Assignment Centre: 3.3.5

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.3.6 Vérifier l'énumération Couleur

Le fonctionnement des classes dans le modèle doit être validé par des tests JUnit.

Travail à faire

Créer, dans le paquetage **fr.eseo.e3.poo.projet.blox.modele** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **fr.eseo.e3.poo.projet.blox.modele.CouleurTest**

Implémenter dans cette classe, les tests JUnit5 permettant de tester les fonctionnalités définies pour l'énumération **fr.eseo.e3.poo.projet.blox.modele.Couleur**.

- Exécuter votre classe **CouleurTest**
- Modifer votre classe de production (**fr.eseo.e3.poo.projet.blox.modele.Couleur**) et de test (**fr.eseo.e3.poo.projet.blox.modele.CouleurTest**) si nécessaire.

Valider votre travail avec l'Assignment Centre: 3.3.6

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.3.7 Créer une représentation visuelle des Pieces

Pour représenter une **Piece** (**Tetromino** ou **Pentomino**) graphiquement, nous utiliserons la classe **VuePiece**. Cette classe sera chargée de dessiner sur le **VuePuits** une représentation de la **Piece**. Pour le faire, nous utiliserons la méthode d'instance **fill3DRect** de la classe **java.awt.Graphics2D** pour chaque **Element** de la **Piece**, en assurant que l'effet 3D semble être surélevé au-dessus de la surface. Nous utiliserons la couleur d'affichage définie dans l'énumération pour le remplissage. Pour aider le joueur, nous allons identifier l'élément de reférence, en utilisant une teinte plus clair pour son remplissage.

La classe **java.awt.Color** nous fourni une méthode **brighter()** qui en théorie nous permet de créer une teinte plus claire. Si on regarde dans la code source de cette classe[7] on voit que cette méthode est écrite comme:

```
private static final double FACTOR = 0.7;
public Color brighter() {
    int r = getRed();
    int g = getGreen();
    int b = getBlue();
    int alpha = getAlpha();

    /* From 2D group:
     * 1. black.brighter() should return grey
     * 2. applying brighter to blue will always return blue, brighter
     * 3. non pure color (non zero rgb) will eventually return white
     */
    int i = (int)(1.0/(1.0-FACTOR));
    if ( r == 0 && g == 0 && b == 0) {
        return new Color(i, i, i, alpha);
    }
    if ( r > 0 && r < i ) r = i;
```

[7] La code source des classes de l'API de Java se trouve généralement dans une archive src.zip dans le repertoire **lib** de l'installation du JDK.

```
        if ( g > 0 && g < i ) g = i;
        if ( b > 0 && b < i ) b = i;

        return new Color(Math.min((int)(r/FACTOR), 255),
                         Math.min((int)(g/FACTOR), 255),
                         Math.min((int)(b/FACTOR), 255),
                         alpha);
    }
```

Le problème avec cette implémentation est que si une des composantes (rouge, vert ou bleu) ont la valeur zero, il ne change pas, par exemple la couleur **`Color.BLUE`** ne change pas (même si le deuxième point dans le commentaire lignes 7 à 11 implique autrement). Nous ne pouvons pas utiliser cette implémentation, et donc nous avons besoin de créer notre propre algorithme pour créer une nouvelle teinte[8]:

1. Définir une constante MULTIPLIER_TEINTE qui sera le facteur de multiplication pour calculer le nouveau teint.
2. Récupérer les valeurs entières (entre 0 et 255) qui représentent les différents composants rouge, vert et bleu de la couleur
3. pour chaque composante c:
   - calculer sa nouvelle valeur

$$c = c + (255 - c) * MULTIPLIER\_TEINTE$$

   - Utiliser la transtypage pour créer une valeur entière à partir de le résultat de ce calcul
4. Créer une nouvelle instance de la classe **`java.awt.Color`** en utilisant les nouvelles valeurs pour les composants rouge, vert et bleu.

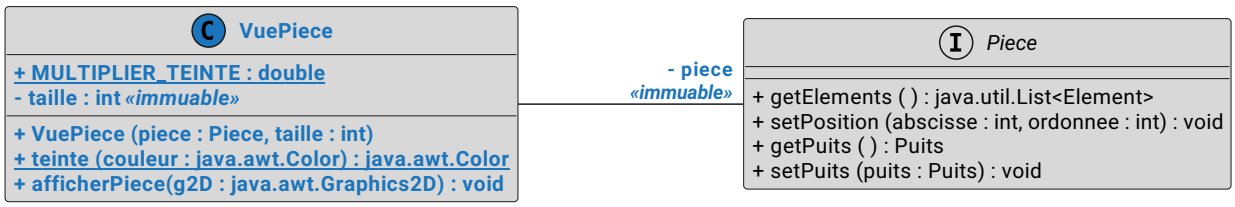


Figure 17: La classe VuePiece

Travail à faire

Écrire la classe **`fr.eseo.e3.poo.projet.blox.vue.VuePiece`** telle qu'elle est représentée dans la figure 17.

- Définir la constante de classe **`MULTIPLIER_TEINTE`** avec une valeur réelle[a]. m:

$$m \in \mathbb{R}, m \in]0.0, 1.0[$$

- Le constructeur doit stocker dans les variables d'instance immuable la **`Piece`** et la taille d'affichage de chaque **`Element`**
- La méthode *de classe* teinte() prend en paramètre une instance de la classe **`java.awt.Color`** et retourne une nouvelle instance de cette classe en utilisant l'algorithme ci-dessus pour sa création.
- dans la méthode **`afficherPiece`** en utilisant l'instance de **`java.awt.Graphics2D`** fourni en paramètre, dessiner en utilisant ses méthodes **`setColor()`** et **`fill3DRect()`** les différents **`Element`**s de la **`Piece`**. Nous devons utiliser la couleur définie pendant la création de la **`Piece`**, en nous assurant d'utiliser pour la première **`Element`** la teinte retournée par la méthode **`teinte()`**.

[8] En théorie de couleur, une teinte est le mélange d'une couleur avec du blanc, ce qui augmente la légèreté

[a] Le choix de cette valeur est libre, et dépend sur vos propres goûts

Valider votre travail avec l'Assignment Centre: 3.3.7

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.3.8 Ajouter les VuePieces dans le VuePuits

Pour l'instant, nous avons redéfini la méthode **`paintComponent()`** dans la classe **`VuePuits`** pour qu'elle affiche une grille. Maintenant, nous voulons ajouter la fonctionnalité pour que la **`VuePiece`** qui représent la piece actuelle du Puits peut être aussi affichées. Pour le faire, nous avons besoin d'associer une VuePiece avec le vuePuits et ensuite modifier la méthode **`VuePiece`** pour qu'elle fasse appel à la méthode **`afficherPiece`** de cette VuePiece.

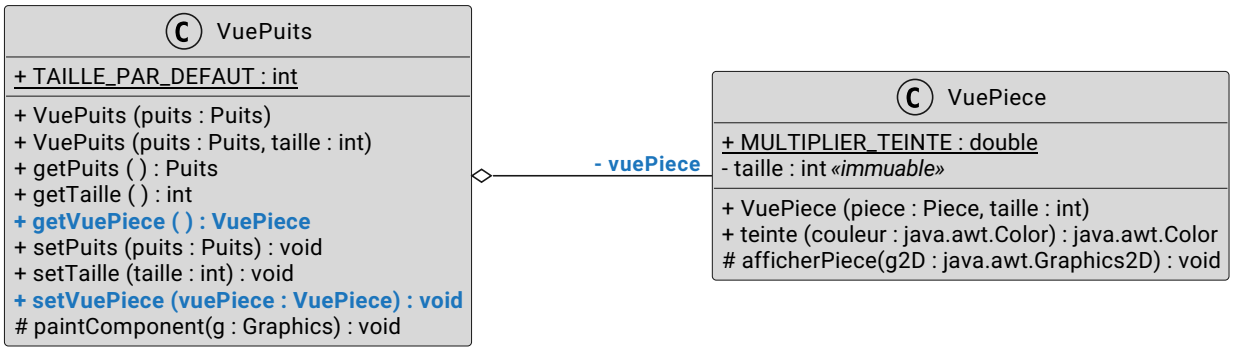


Figure 18: Le VuePuits et la VuePiece

Travail à faire

Modifier la classe **`fr.eseo.e3.poo.projet.blox.vue.VuePuits`** telle qu'elle est représentée dans la figure 18.

- Créer l'association entre VuePuits et VuePiece avec l'accesseur et mutateur pour ce dernier.
- Faire en sorte qu'après la construction, il n'y a aucune VuePiece associé avec le VuePuits.
- Modifier la méthode **`paintComponent()`** pour qu'elle dessine la VuePiece associée une fois que la grille est affichée (seulement s'il y en a une associée).

Valider votre travail avec l'Assignment Centre: 3.3.8

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.3.9 Vérifier l'affichage des VuePiece

Travail à faire

Modifier, dans le paquetage **`fr.eseo.e3.poo.projet.blox.vue`** qui se trouve dans le dossier source `test`, la classe suivante:

- **`fr.eseo.e3.poo.projet.blox.vue.VuePuitsAffichageTest`**
- Ajouter les lignes de code necessaire, pour que vous utilisiez la classe **`UsineDePiece`**, pour ajouter une **`Piece`** dans votre **`Puits`**. Ensuite ajouter une **`VuePiece`** pour afficher la Pièce dans le **`VuePuits`**.
- Lancer l'application de test et vérifier manuellement que deux fenêtres s'affichent

sur l'écran avec les bons titres et les bonnes dimensions et que dans chaque fenêtre, une grille de couleur gris clair s'affiche sur un fond blanc.

- Modifer vos classes de production (**`fr.eseo.e3.poo.projet.blox.vue.VuePuits`** et **`fr.eseo.e3.poo.projet.blox.vue.VuePiece`**)
  et de test (**`fr.eseo.e3.poo.projet.blox.vue.VuePuitsAffichageTest`**)
  si necessaire.

Valider votre travail avec l'Assignment Centre: 3.3.9

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

Attention, ne touchez pas le clavier ni la souris pendant l'exécution de tests par l'Assignment Centre. Lors des tests de l'interface graphique, l'Assignment Centre affiche les fenêtres sur l'écran et peut interagir avec. Si vous manipulez le clavier ou la souris pendant ces tests, l'Assignment Centre peut enregistrer des erreurs.

3.3.10 Modifier VuePiece automatiquement quand une nouvelle Piece est ajouté au Puits

Avec le code que nous avons écrit, nous pouvons créer un **`Puits`**, ajouter les **`Piece`**s qui ont été construit par notre **`UsineDePiece`**. Notre code voius permet aussi de manuellement ajouter une représentation visuelle (une **`VuePiece`**) de cette **`Piece`** dans la fenêtre qui contient le **`VuePuits`**.

Avec notre implémentation, nous sommes obligés de manuellement modifier la **`VuePiece`** chaque fois que nous changeons la pieceActuelle de notre **`Puits`**. Une solution de ce problème est d'ajouter le code dans la classe **`Puits`** pour faire ce changement. Mais, même si la solution est simple, il y a un autre problème avec cette solution.

En effet, dans la section 2, nous avons vu qu'*idéalement un objet Modèle n'a pas de connexion avec l'interface utilisateur permettant de le visualiser…*. Pour ce projet, nous avons choisi d'utiliser la bibliotheque swing pour l'interface graphique, mais rien nous empeche de toujours utiliser swing, et dans l'avenir, nous pouvons changer pour utiliser JavaFX, ou une autre bibiliotheque (lwjgl[9], …). Si nous ajoutons le code de modifier notre **`VuePuits`** directement dans le code modèle, ces changements seront plus difficiles à implémenter.

Une autre solution (ce que nous utiliserons) est d'ajouter la fonctionnalité qui permet des instances des classes de s'enregistrer en tant qu'un objet qui veut recevoir les informations chaque fois qu'une propriété d'une instance d'une autre classe est modifié. Avec cette façon de procéder, l'objet modèle a besoin d'être modifié pour que d'autres classes puisse s'enregistrer d'écouter pour ces changements. L'objet modèle doit aussi être modifié pour qu'il puisse générer les notifications quand un changement a été fait. Les classes qui s'enregistrent doit, elles aussi, être modifiées pour que chaque fois qu'elles écoutent une notification, qu'elles peuvent modifier leur comportement.

Pour notre projet, nous allons faire en sort que la classe **`Puits`** permet d'autres classes de s'enregistrer pour être informées quand les pieceActuelle et pieceSuivante sont modifiées. Nous allons modifier la classe **`VuePuits`** pour qu'elle puisse écouter pour ces changements, et que chaque fois que la pieceActuelle change, elle modifie la VuePiece pour qu'elle représente toujours la pieceActuelle.

Pour implémenter cette fonctionnalité, nous allons utiliser les classes / interfaces qui se trouvent dans l'API de Java: **`java.beans.PropertyChangeSupport`** et **`java.beans.PropertyChangeListener`**. La classe PropertyChangeSupport nous permet d'enregistrer (ou dénregistrer) les classes qui écoutent les changements, et d'assurer que chaque classe qui écoute pour les changments recoivent bien les notifications. L'interface
`PropertyChangeListener` est utilisé pour assurer que notre **`VuePuits`** respect le contrat necessaire pour écouter ces notifications. Ce sera possible que nous implémentions le même fonctionnalité sans utiliser ces classes, par contre, c'est beaucoup plus simple de les utiliser.

[9] Lightweight Java Game Library - qui founi un API en Java pour utiliser OpenGL

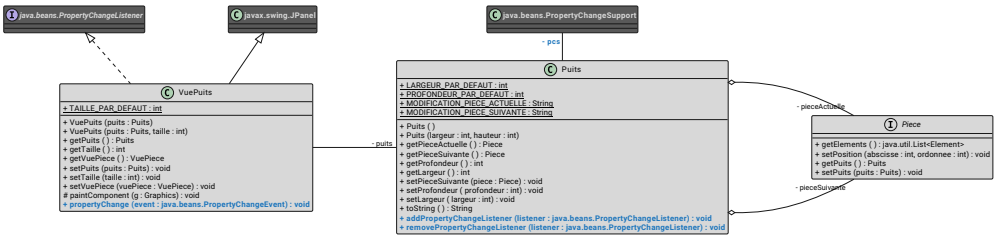


Figure 19: Les modifications pour Puits et VuePuits

Travail à faire

Modifier la classe **`Puits`** telle qu'elle est représentées dans la figure 19. Le diagramme de séquence dans la figure 20 peut vous aider implémenter la fonctionnalité de cette section.

- Assurer qu'après construction, la variable d'instance **`pcs`** est initialisé en passant l'instance de Puits comme paramètre vers le constructeur de **`PropertyChangeSupport`**.
- Créer les constants de classe **`MODIFICATION_PIECE_ACTUELLE`** et **`MODIFICATION_PIECE_SUIVANTE`** pour qu'elles contiennent des chaînes de caractères qui peuvent être utilisé pour identifier quelle pièce a été modifiée.
- Ajouter les méthodes **`addPropertyChangeListener()`** et **`removePropertyChangeListener()`**. Ces deux méthodes doivent appeler les méthodes équivalentes pour la variable d'instance **`pcs`**[a].
- Modfier la méthode **`setPieceSuivante`** pour que chaque fois que soit la pieceSuivante, soit la pieceActuelle est modifiée, que la méthode **`firePropertyChange(String nomPropriete, Object ancienValeur, Object nouvelleValeur)`** est bien appelé.
  - Le premier argument doit être choisi entre un des deux constants de classe défini ci-dessus
  - Le deuxième argument doit être l'ancienne valeur de pieceActuelle ou de pieceSuivante
  - Le dernier argument doit être la nouvelle valeur de pieceActuelle ou de pieceSuivante

Modifier la classe **`VuePuits`** telle qu'elle est représentée dans la figure 19.

- La classe doit implémenter l'interface **`PropertyChangeListener`**.
- Ajouter la méthode **`propertyChange()`** défini dans le contrat de **`PropertyChangeListener`**. Dans cette méthode, nous devons vérifier que c'est bien un changement de la pieceActuelle, si c'est le cas, nous appelons la méthode setVuePiece pour que la nouvelle valeur de pieceActuelle est utilisé.
- Modifier la visibilité de la méthode **`setVuePiece()`** pour qu'elle est privé[b]. Ce changement est fait pour renforcer que la VuePiece sera changer seulement quand la classe VuePuits écoute un changement.
- Vous devez assurer que quand le Puits associé avec le VuePuits change (un appel vers **`setPuits()`** que ce VuePuits n'est plus associé en tant que **`PropertyChangeListener`** avec l'ancien Puits, mais seulement avec le nouveu Puits.

[a] en effet, nous écrivons ces deux méthodes pour simplifier l'ajout (ou la suppression) des listeners.
[b] Attention, ce changement va forcément casser votre code de test dans **`VuePuitsAffichageTest`**.

Valider votre travail avec l'Assignment Centre: 3.3.10

L'Assignment Centre vérifie que la structure de votre classe est correcte.

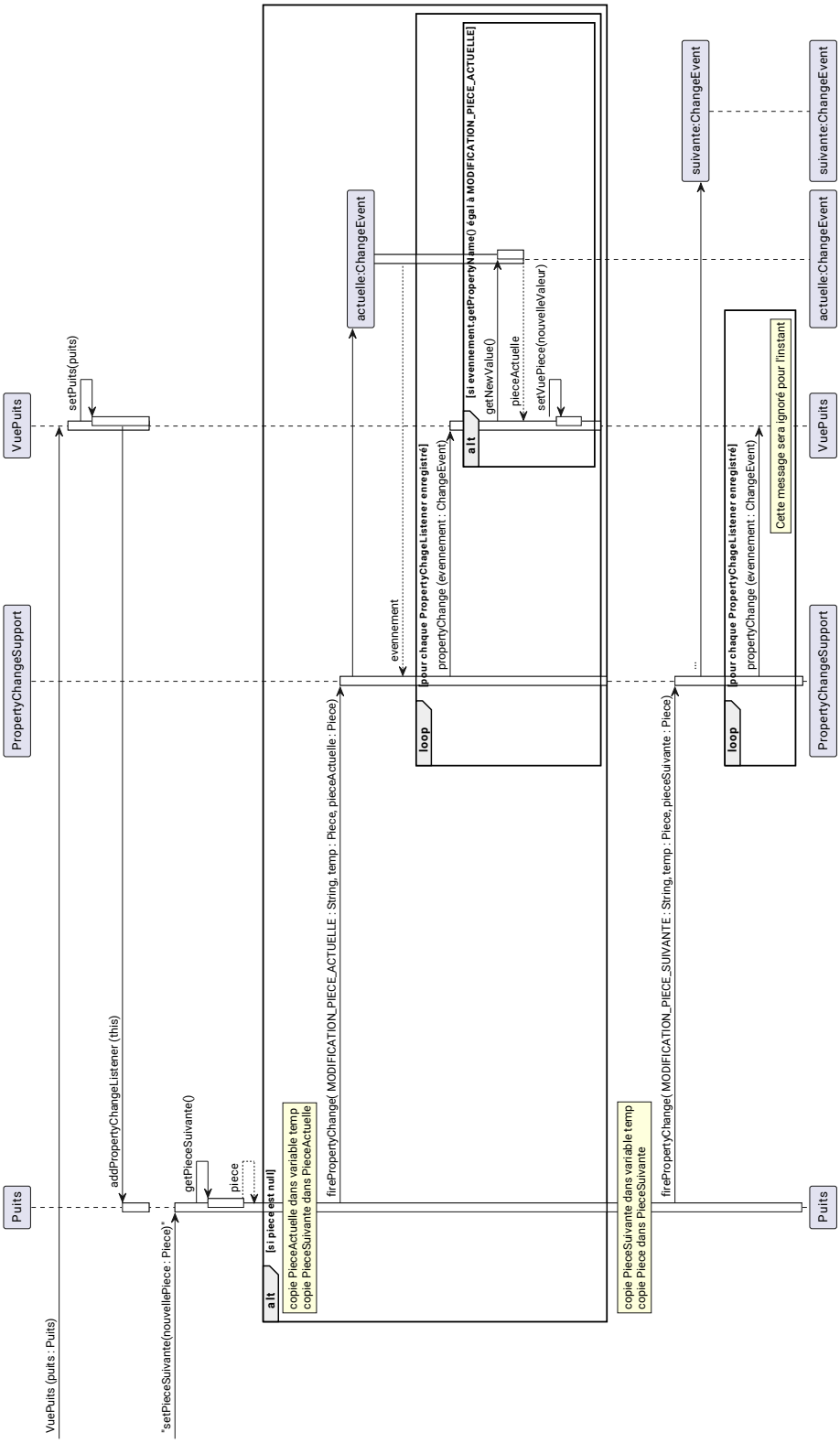


Figure 20: Diagramme de sequence modification d'une piece

3.3.11 Vérifier mise à jour automatique de VuePiece

Travail à faire

Modifier votre classe de test
`fr.eseo.e3.poo.projet.blox.vue.VuePuitsAffichageTest`

- Supprimer les appels éventuels de setVuePiece()
- Faire en sorte qu'après construction de votre **`VuePuits`** et avant les appels de **`setPieceSuivante()`**, que votre instance de VuePuits est enregistré comme listener de votre instance de Puits.

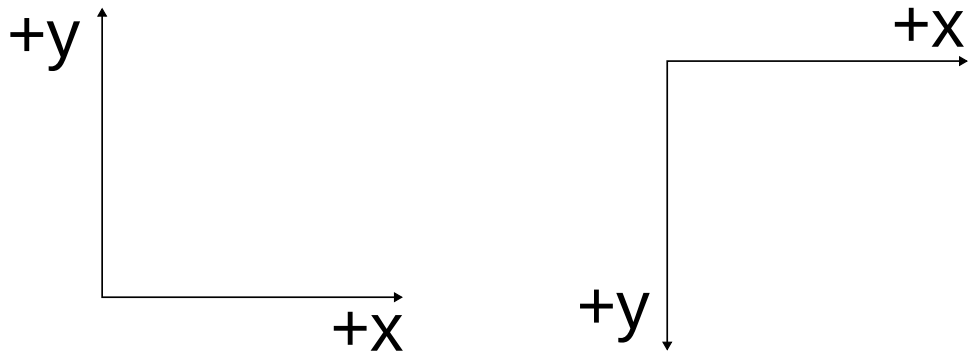
Valider votre travail avec l'Assignment Centre: **3.3.11**

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

Attention, ne touchez pas le clavier ni la souris pendant l'exécution de tests par l'Assignment Centre. Lors des tests de l'interface graphique, l'Assignment Centre affiche les fenêtres sur l'écran et peut interagir avec. Si vous manipulez le clavier ou la souris pendant ces tests, l'Assignment Centre peut enregistrer des erreurs.

3.4 Ajouter la fonctionnalité à la logique du jeu

Maintenant qu'il est possible de visualiser le **`Puits`** et la **`Piece`** actuelle de manière graphique, il est nécessaire d'ajouter les fonctionnalités permettant à une **`Piece`** de bouger en fonction des ordres de l'utilisateur (ordre de translation horizontale ou vers le bas, ou un ordre de rotation). Dans un premier temps il ne sera pas nécessaire de nous préoccuper des cas où la **`Piece`** sort deslimites du **`Puits`** lors d'une translation / rotation.



(a) Repère standard (b) Repère informatique

Figure 21: Deux types de repères

Important!

Repère informatique: en règle générale, l'origine d'un repère cartésien orthonormé (figure 21a) se situe dans le coin inférieur gauche du repère. L'abscisse d'un point croît lorsqu'on se déplace vers la droite le long de l'axe horizontal (l'axe des x). L'ordonnée d'un point croît lorsqu'on se déplace vers le haut le long de l'axe vertical (l'axe des y).
Les repères utilisés par les systèmes informatiques (figure 21b) sont différents dans la mesure où leur origine se situe en haut à gauche du repère, l'abscisse d'un point croît lorsqu'on se déplace vers la droite. Cependant, l'ordonnée d'un point croît lorsque l'on se déplace vers le bas le long de l'axe vertical.
Compte tenu de l'orientation non traditionnel de ce repère, les matrices de rotations et les vecteurs de déplacement ne fonctionnent pas telles quelles et doivent être adaptées.

3.4.1 Faire déplacer les Pieces

Les **Pieces** du jeu peuvent se déplacer selon trois directions:

- horizontale vers la gauche
- horizontale vers la droite
- verticale ver le bas

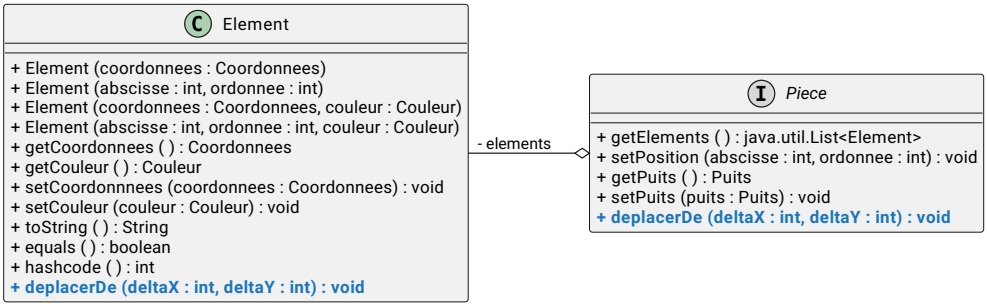


Figure 22: Ajouter la mouvement dans Element et Piece

Étant donné qu'une **Piece** est constituées d'**Element**s, à chaque fois qu'une **Piece** se déplace, ses **Element**s constitutifs doivent être déplaces dans la même direction. Le déplacement doit donc être implémenté, à la fois, dans la classe **Piece** et dans la classe **Element**. Ces méthodes prendront les composantes $deltax$ et $deltay$ du vecteur de déplacement en argument. La méthode de déplacement pour la classe **Piece** lèvera une exception, de type **IllegalArgumentException**, lorsque son argument n'est pas valide (s'il est supérieur à 1 où correspond à un déplacement vers le haut. Nous ne nous préoccuperons pas pour l'instant des éventuelles collisions qui pourraient survenir.

Travail à faire

Modifier la classe **Element** pour qu'elle implémente les modifications représentées dans la figure 22.

- Ajouter la méthode **deplacerDe(int, int)**. Cette méthode doit modifier l'abscisse et l'ordonnée de l'**Element** par le vecteur de déplacement qui a les composantses $deltax$ et $deltay$ en argument.

Modifier l'interface **Piece** pour qu'elle implémente les modifications représentées dans la figure 22.

Modifier la classe abstraite **Tetromino** et / ou une ou plusieurs de ces sous-classes pour que la fonctionnalité décrite dans la section 3.4.1 est implémentée.

- Ajouter la méthode **deplacerDe(int, int)**. La déclaration de cette méthode doit indiquer qu'elle est susceptible de lever une exception de type **IllegalArgumentException**.
- La méthode devra s'assurer que le vecteur de déplacement représente une direction de déplacement valide(vers la gauche, la droite ou le bas).
  - si le déplacement est valide, le déplacement de **Piece** se faire en appelant la méthode **deplacerDe** de chaque **Element** constituent.
  - si le déplacement n'est pas valide, la méthode devra lever une exception comportant un message expliquant la raison de la levée d'exception.

Valider votre travail avec l'Assignment Centre: **3.4.1**

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.4.2 Vérifier la mouvement des Pieces

Travail à faire

Modifier, dans le dossier source test, le cas de test JUnit 5 suivant:

- **`fr.eseo.e3.poo.projet.blox.modele.ElementTest`**
- **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos.ITetrominoTest`**
- **`fr.eseo.e3.poo.projet.blox.modele.pieces.tetrominos.OTetrominoTest`**

Implémenter dans ces classes, les tests JUnit5 permettant de tester les fonctionnalités définies pour les classes **`fr.eseo.e3.poo.projet.blox.modele.pieces.Piece`**, **`fr.eseo.e3.poo.projet.blox.modele.pieces.Tetromino`** (si nécessaire ses sous-classes aussi) et **`fr.eseo.e3.poo.projet.blox.modele.Element`**.

- Exécuter vos classes **`ElementTest`**, **`ITetrominoTest`** et **`OTetrominoTest`**
- Si nécessaire, modifier vos classes de production et / ou de tests.

Attention, les tests doivent valider les deux cas: Un vecteur de déplacement valide ou invalide.

Valider votre travail avec l'Assignment Centre: 3.4.2

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.4.3 Faire tourner les Pieces

Dans le règles du jeu, une **`Piece`** peut tourner par 90° autours de son **`Element`** de référence telle qu'il est représenté dans la figure 1. Une telle rotation peut se faire soit dans le sens horaire soit dans le sens anti-horaire.

Important!

La rotation est un concept simple à comprendre. Cependant, dans le jeu de FallingBlox il est necessaire de faire attention à certains détails:

Rotation autours de l'origine: en règle générale, la rotation se fait autours de l'origine $(0, 0)$; dans le jeu de FallingBlox, elle se fait autour de l'**`Element`** de référence de la **`Piece`**. Ainsi, pour obtenir la rotation de la **`Piece`**, un processus en trois étapes peut être mis en oeuvre:

1. Translater les **`Element`**s de la **`Piece`** d'un vecteur (dx, dy) afin que l'**`Element`** de référence de la **`Piece`** se trouve à l'origine du repère.
2. Effectuer la rotation des **`Element`**s de la **`Piece`** (sauf l'**`Element`** de référence) avec l'origine du repère comme centre de la rotation
3. Translater les **`Element`**s de la **`Piece`** d'un vecteur $(-dx, -dy)$ afin que l'**`Element`** de référence de la **`Piece`** retourne à sa place initiale.

Les OTetrominos: Les **`OTetrominos`** sont différentes des autres **`Pieces`** dans la mesure où il ne peut pas leur être appliquée de rotation.

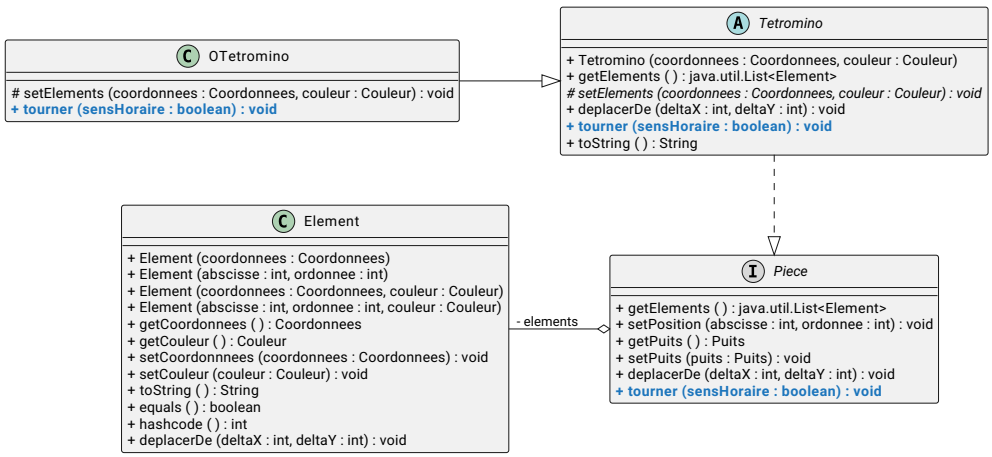


Figure 23: Faire tourner une Piece

Travail à faire

Déclarer la méthode tourner() dans la classe **Piece** et l'implémenter dans **Tetromino** comme elle est définie dans la figure 23.

- en fonction de la valeur du paramètre **boolean**, les **Element**s de la **Piece** devront avoir subit une rotation autour de l'**Element** de référence dans le sens horaire (si le **boolean** vaut **true**) ou anti-horiare (si le **boolean** vaut **false**.

Redéfinir la méthode **tourner()** dans la classe **OTetromino**, de sorte qu'une **OTetromino** ne puisse pas subir de rotation.

Valider votre travail avec l'Assignment Centre: 3.4.3

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.4.4 Vérifier la rotation des Pieces

Travail à faire

Modifier, dans les sous-paquetages de **fr.eseo.e3.poo.projet.blox.modele** qui se trouve dans le dossier source test, les cas de test JUnit 5.
Implémenter dans ces classes, les tests JUnit5 permettant de tester les fonctionnalités définies pour tourner des différents **Pieces**.

- Exécuter vos classes **ITetrominoTest** et **OTetrominoTest**
- Modifier vos classes de production et test si nécessaire.

Attention, les tests doivent valider les deux cas: Un vecteur de déplacement valide ou invalide.

Valider votre travail avec l'Assignment Centre: 3.4.4

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

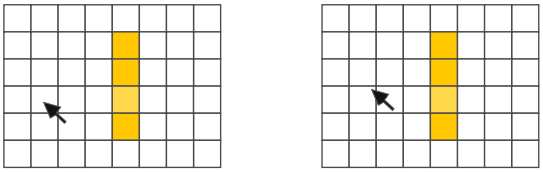
3.5 Ajouter les interactions utilisateur

Le joueur doit pouvoir interagir avec le jeu. Dans un premier temps, nous nous contenterons sur ses possibilités de contrôler le déplacement et la rotation de la pieceActuelle grâce à la souris. Ce n'est probablement pas le moyen le plus simple pour le joueur de contrôler les déplacements d'une **Piece**, mais *pédagogiquement*, cette méthode à deux avantages:

- elle permet d'illustrer le fait qu'une souris génère un nombre considérable d'événements

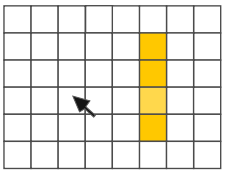
- elle ne requiert pas que nous nous préoccupons du système de *focus* (qui pourra être un sujet intéressant pour les extensions à apporter au jeu, voir la section 4)

3.5.1 Utiliser la souris pour bouger la pièce horizontalement



(a) Début de mouvement du souris

(b) souris a bougé, mais toujours dans la même colonne



(c) souris a changé la colonne, la pièce bouge

Figure 24: Déplacement avec la souris. Nous remarquerons que la Pièce ne se déplace que lorsque le pointer de la souris change de colonne.

Pour les déplacements horizontaux, nous ferons en sorte qu'un déplacement de la souris vers la gauche déplace la **`Piece`** vers la gauche et un mouvement de la souris vers la droite la déplace vers la droite. Pour obtenir ce résultat, il sera nécessaire de créer une classe qui implémente l'interface **`java.awt.event.MouseMotionListener`**. Cette classe devra ensuite être inscrite auprès du **`VuePuits`** en tant que **`MouseMotionListener`**. L'interface **`MouseMotionListener`** requiert que les classes, qui l'implémentent, définissent deux méthodes **`mouseMoved()`** et **`mouseDragged`**. À terme, il sera donc nécessaire d'implémenter ces deux méthodes, mais dans un premier temps, nous ne nous concentrerons que sur la méthode **`mouseMoved`**.

Un événement de type **`MouseMoved`** est créé à chaque fois que l'utilisateur déplace la souris. En fait, chaque *micro-déplacement* de la souris crée un événement. Ceci implique que même un petit déplacement provoque la création d'un très grand nombre d'événements. Dans notre cas, il n'est donc pas question de déplacer la **`Piece`** dans le **`Puits`** à chaque fois qu'un événement est créé. En effet, ça rendrait le jeu injouable puisqu'un très petit déplacement de la souris provoquerait un déplacement latéral de la **`Piece`** de plusieurs colonnes sur le **`Puits`**. Nous choisirons de déplacer la **`Piece`** dans le **`Puits`** uniquement lorsque le pointeur de la souris passe d'une colonne à sa voisine sur la **`VuePuits`** (idéalement lorsque le pointeur traverse la ligne verticale qui sépare deux colonnes, voir figure 24

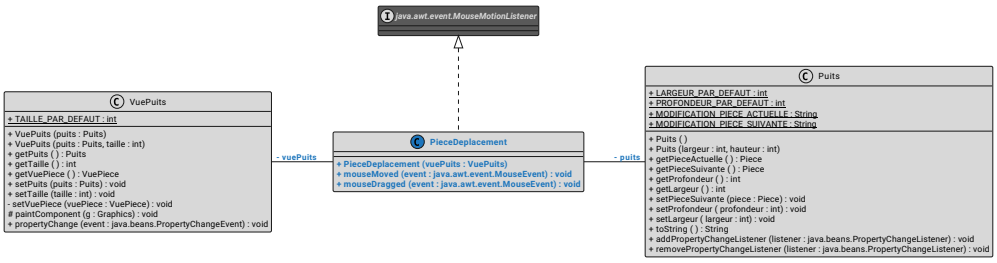


Figure 25: La classse PieceDeplacement

Travail à faire

Créer la classe **`fr.eseo.e3.poo.projet.blox.controleur.PieceDeplacement`** telle qu'elle apparaît dans la figure 25. Cette classe devra implémenter l'interface **`java.awt.event.MouseMotionListener`** interface.

- Écrire le constructeur chargé d'implémenter l'association entre **PieceDeplacement** et les classes **VuePuits** et **Puits**.
- Redéfinir la méthode **mouseMoved**:
  - seulement s'il y a une pieceActuelle définie pour le **Puits**:
  - si c'est la première fois qu'un tel événement a été créé, stocker le numéro de la colonne au-dessous de celle où se trouve le pointeur de la souris, sinon
    * vérifier le numéro de la colonne au-dessous de laquelle se trouve le pointeur de la souris et le comparer avec le numéro de la dernière colonne au-dessous de laquelle elle se trouvait.
    * si la souris a changé de colonne, alors la pieceActuelle du **Puits** peut être déplacé si, toutefois, le déplacement ne lève pas d'exceptions.
    * stocker le numéro de la colonne au-dessous de celle où se trouve le pointeur de la souris
- Ajouter les autres méthodes manquantes[a].

Modifier la classe **VuePuits** afin que;

- dans son/ses constructeur(s) une instance de **PieceDeplacement** soit ajoutée à la liste de ses gestionnaires d'événements de type **MouseMotionListener**
- il y a toujours qu'une seule instance de **PieceDeplacement** associé comme **MouseMotionListener** et que l'information sur le **Puits** et **VuePuits** est toujours à jour dans l'instance de la classe **PieceDeplacement**.
  - Pour le faire, chaque fois que la méthode **setPuits()** dans **VuePuits** est appelé, l'instance de PieceDeplacement doit être modifié - *il y a plusieurs solutions qui peuvent fonctionner ici*.

[a] Utiliser la fonctionnalité de l'ide pour ajouter les méthodes manquantes

Valider votre travail avec l'Assignment Centre: 3.5.1

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.5.2 Vérifier le mouvement horizontale avec la souris

Travail à faire

Créer, dans le paquetage **fr.eseo.e3.poo.projet.blox.controleur** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **fr.eseo.e3.poo.projet.blox.modele.pieces.PieceDeplacementTest**

Implémenter dans cette classe, une application qui crée une fenêtre avec une représentation visuelle d'un **Puits**. Faire en sorte qu'une **Piece** soit bien ajouté quelque part dans le **Puits** et que sa visualisation graphique (une **VuePuits**) soit visible. Vérifier que les mouvements de la souris permettent de déplacer correctement la pieceActuelle dans l'axe horizontal.

Valider votre travail avec l'Assignment Centre: 3.5.2

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.5.3 Utiliser la souris pour bouger la pièce verticalement

Parfois, l'utilisateur sait exactement où il veut placer une **Piece**. Dans ce cas, il est intéressant de lui fournir un moyen d'accélérer la chute de la **Piece** pour pouvoir passer plus rapidement à la pièce suivante. Nous permettrons à l'utilisateur d'accélérer la chute de la **Piece** courante en tournant la roulette de sa souris vers lui. Pour obtenir ce résultat, il est nécessaire de modifier les classes **PieceDeplacement** et **VuePuits** pour qu'il puisse y avoir une réaction lors d'un mouvement de la roulette (**MouseWheel**). Pour utiliser la

fonctionnalité pour gérer les interactions avec la souris, la classe **PieceDeplacement** doit implémenter l'interface **MouseWheelListener**.

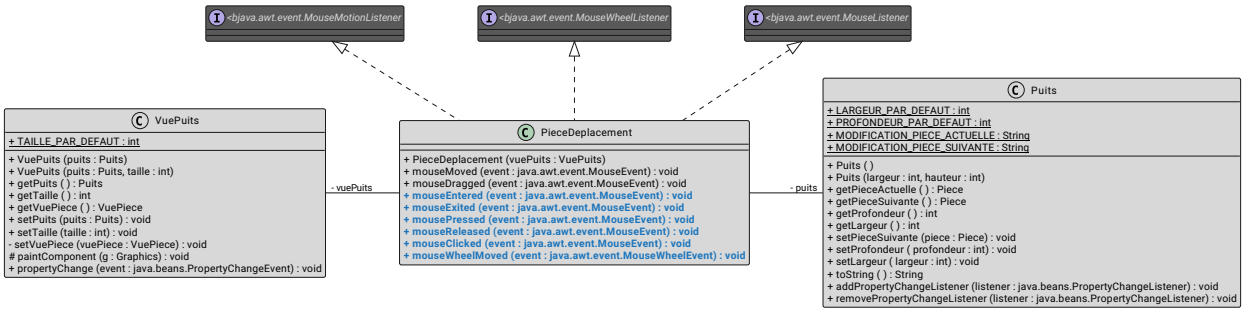


Figure 26: PieceDeplacement implements

Avant d'implémenter cette fonctionnalité, nous avons besoin de corriger un problème potentiel. Lors des déplacements de la souris, il est possible que le pointeur de la souris sorte de la fenêtre. Dans ce cas-là, avec l'implémentation que nous avons fait, c'est possible que si le pointeur rentre dans la fenêtre dans une autre colonne, la pièce va se déplacer, même si ce n'est pas voulu. Pour résoudre ce problème, nous pouvons réinitialiser la dernière colonne chaque fois que la souris rentre dans la fenêtre. L'interface **MouseListener** fournie les méthodes qui permettent de gérer les événements pour entrer ou pour sortir d'une fenêtre.

Pour récapituler, nous avons besoin d'implémenter **MouseListener**, **MouseMotionListener** et **MouseWheelListener**. L'API de java nous fournit une classe **MouseAdapter** qui implémente déjà ces interfaces. C'est une classe qui contient des méthodes avec aucune fonctionnalité dedans, nous pouvons faire en sorte que notre classe soit une sous-classe de **MouseAdapter** et pas une implémentation des trois interfaces. L'utilisation de la classe **MouseAdapter** nous permet de produire un code qui est plus facile à lire, car nous avons juste besoin d'implementer les méthodes que nous utilisons, et pas toutes les méthodes définit dans les interfaces (un total de huit méthodes, en sachant que nous utilisons que trois).

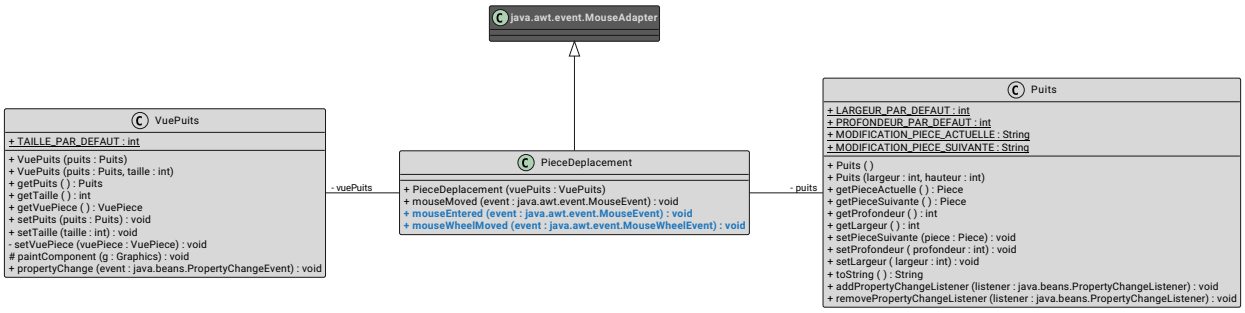


Figure 27: PieceDeplacement extends

Travail à faire

Modifier la classe **PieceDeplacement** telle qu'elle est représentée dans soit la figure 27 soit la figure 26. Dans les deux cas, implémenter la fonctionnalité suivante:

- Dans la méthode **mouseEntered()** faire en sorte que chaque fois que la souris rentre dans la fenêtre, que la dernière colonne soit reinitialisée - pour que la pièce ne déplace pas avant que le pointeur de la souris changent encore la colonne.
- Dans la méthode **mouseWheelMoved()**, après avoir bien vérifié qu'il y a une pieceActuelle définie pour le **Puits**, déplacer la **Piece** vers le bas si la roulette de la souris a été tournée vers l'utilisateur (**getWheelRotation()** est supérieur à 0).

Modifier la classe **VuePuits** pour que pendant le processus de construction, la classe **PieceDeplacement** soit enregistré en tant qu'un **MouseListener**, **MouseMotionListener** et **MouseWheelListener**.

- Vérifier bien qu'il y a toujours qu'une seule instance de **PieceDeplacement** enregistré avec le **VuePuits** (*même après un appel de* **setPuits()**.

- Vérifier que c'est toujours le même instance de **PieceDeplacement** qui est utilisé pour le **MouseListener**, **MouseWheelListener** et **MouseMotionListener**.

Valider votre travail avec l'Assignment Centre: 3.5.3

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.5.4 Vérifier le mouvement vertical avec la souris

Travail à faire

Modifier (si besoin), dans le paquetage **fr.eseo.e3.poo.projet.blox.controleur** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **fr.eseo.e3.poo.projet.blox.modele.pieces.PieceDeplacementTest**

Implémenter dans cette classe, une application qui crée une fenêtre avec une représentation visuelle d'un **Puits**. Faire en sorte qu'une **Piece** soit bien ajoutée quelque part dans le **Puits** et que sa visualisation graphique (une **VuePuits**) est visible. Vérifier que les mouvements de la souris déplacent correctement la pieceActuelle dans l'axe horizontale, et que quand l'utilisateur tourne la roulette, la pieceActuelle descend le **Puits**.

Valider votre travail avec l'Assignment Centre: 3.5.4

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.5.5 Utiliser la souris pour faire tourner la piece

Tout comme le joueur peut contrôler le mouvement de la **Piece** actuelle avec sa souris, il doit pouvoir utiliser sa souris pourfaire tourner la **Piece** lors de sa chute: le bouton droit de la souris la fera tourner dans le sens horaire, tandis que le bouton gauche lui fera subir une rotation dans le sens anti-horaire. Pour pouvoir utiliser les boutons de la souris pour contrôler les actions de rotation, nous devons créer la classe **PieceRotation** qui devra implémenter l'interface **java.awt.MouseListener** et être ajoutée à la liste de gestionnaires d'événements souris du **VuePuits**.

Le bouton utilisé lors d'un événement souris, peur être idenifiés en utilisant la méthode **getButton()** qui renvoie un entier pour designer le bouton. Cette valeur peut être égale à un des constants définis dans la classe **MouseEvent**:

- **NOBUTTON**
- **BUTTON1**
- **BUTTON2**
- **BUTTON3**

Le problème est quel bouton est représenté avec quel constant (pour une souris avec que deux boutons, trois boutons ou de plus…).

La classe **SwingUtilities** nous fournisse des méthodes qui nous aident de plus facilement identifier les boutons. Les méthodes suivantes permettent de vérifier quel bouton a été utilisé:

- **SwingUtilities.isLeftMouseButton(MouseEvent)**
- **SwingUtilities.isRightMouseButton(MouseEVent)**
- **SwingUtilities.isMiddleMouseButton(MouseEVent)**

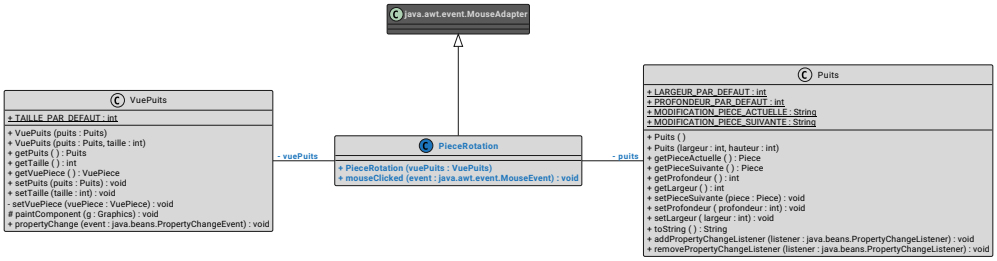


Figure 28: La classe PieceRotation

Travail à faire

Créer la classe **`fr.eseo.e3.poo.projet.blox.controlleur.PieceRotation`** telle qu'elle est représentée dans la figure 28

- Redéfinir la méthode **`mouseClicked`** de telle sorte à ce qu'elle permette au joueur de faire tourner la pièce de 90° dans le sens horaire en appuyant sue le bouton droit de la souris et du même angle dans le sens anti-horaire lorsqu'il appuie sur le bouton gauche. La méthode devra tenir compte de tous les comportements exceptionnels qui peuvent subvenir lors de la rotation.

Modifier la classe **`VuePuits`** pour que pendant le processus de construction, la classe **`PieceRotation`** soit enregistré en tant qu'un **`MouseListener`**.

- Vérifier bien qu'il y a toujours qu'une seule instance de **`PieceRotation`** enregistré avec le **`VuePuits`** (*même après un appel de* **`setPuits( )`**.
- Vérifier que c'est toujours le même instance de **`PieceDeplacement`** qui est utilisé pour le **`MouseListener`**, **`MouseWheelListener`** et **`MouseMotionListener`**.

Valider votre travail avec l'Assignment Centre: 3.5.5

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.5.6 Vérifier la rotation avec la souris

Travail à faire

Créer, dans le paquetage **`fr.eseo.e3.poo.projet.blox.controleur`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`fr.eseo.e3.poo.projet.blox.modele.pieces.PieceRotationTest`**

Implémenter dans cette classe, une application qui crée un fenêtre avec une représentation visuelle d'un **`Puits`**. Faire en sorte qu'une **`Piece`** est bien ajouté quelque part dans le **`Puits`** et que sa visualisation graphique (une **`VuePuits`**) est visibile. Vérifier que quand l'utilisateur fait un clic avec soit le bouton gauche, soit le bouton droit de la souris, la piece-Actuelle tourne correctement.

Valider votre travail avec l'Assignment Centre: 3.5.6

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.6 Ajouter le Tas et refactorisation de la logique du jeu

3.6.1 Ajouter le tas en bas du Puits

La **`Piece`** qui se déplace et tourne sur le **`Puits`**, n'est pas le seul contenu du **`Puits`**. Lorsqu'une **`Piece`** atteint le fond du **`Puits`**, la **`Piece`** se disloque et ses **`Element`**s sont ajoutés à un **`Tas`**d'**`Element`**s. L'objectif du jeu est de constituer des lignes d'**`Element`**s horizontales complètes au fond du **`Puits`** afin de pouvoir les retirer du **`Tas`**. Lorsqu'une **`Piece`** atteint une position où elle ne peut plus se déplacer vers le

bas (si après le déplacement, au moins un de ses **Element**s chevaucherait un autre **Element** de la **Pile**, alors les **Element**s de la **Piece** doivent être ajoutée au **Tas** et une nouvelle **Piece** (la pieceSuivante) doit commencer à chuter depuis le haut du **Puits**.

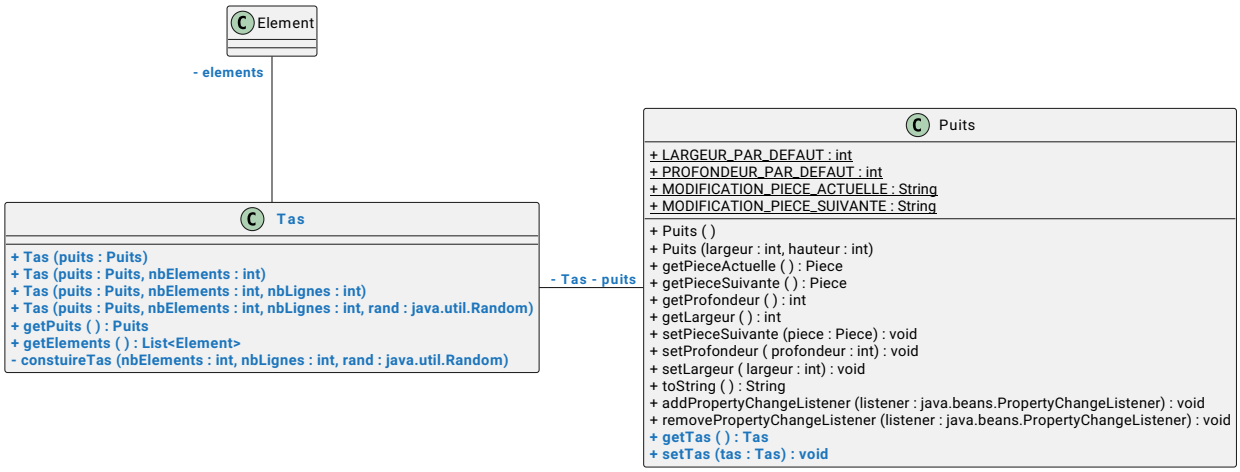


Figure 29: Intégration de la classe Tas

Dans cette section, nous ne nous préoccuperons pas de vérifier que la **Piece** touche le **Tas**, ou qu'une ligne complète d'**Element**s a été formée dans le **Tas**. Nous nous contenterons seulement de créer le **Tas**. Les **Element**s dans le **Tas** seront sauvegardés dans une **List** des **Element**s. Cette structure de données permet qu'on n'utilise que l'espace en mémoire nécessaire pour stocker les **Element**s dans le **Tas** et de faciliter l'ajoute ou la suppression des **Element**s.

Travail à faire

Implémenter la classe **Tas** telle qu'elle apparaît dans la figure 29.

- Les constructeurs devront créer les liens entre les classes **Tas**, **Puits** et **Element**. Ils devront aussi créer une instance d'un **java.util.ArrayList<Element>** et le stocker dans un variable d'instance de type **java.util.List<Element>**.
- Le constructeur par défaut créer un Tas vide.
- Chaque constructeur doit appeler la méthode **construireTas** pour répartir aléatoirement **nbElements** dans **nbLignes** d'en bas du **Puits**. Le dernier paramètre de la méthode est une instance de la classe Random, qui sera utilisé pour générer les valeurs aléatoires. La méthode **construireTas** **doit implémenter exactement**, l'algorithme de la figure 30.
- Le constructeur à quatre arguments, **Tas(Puits, int, int, Random)** permet de passer une instance de la classe **Random** avec un *seed* connu - pour permettre de créer toujours le même distribution des **Element**s dans le **Tas**.
- Le constructeur à trois arguments doit appeler la méthode construireTas avec une instance de Random, avec un **seed** quelconque.
- Le constructeur à deux arguments, fait le même chose que pour le constructeur aux trois arguments, sauf que le nombre de lignes à utiliser (n) est calculé selon la formule: $n = nbElements \div largeurDuPuits + 1$).
- Le constructeur avec qu'un seul argument doit créer le Tas avec aucune **Element** dans la **List**.
- Le constructeur à trois arguments.
- Les constructeurs et la méthode **construireTas()** doivent lancer une exception de type **IllegalArgumentException** si le nombre d'**Element**s sera trop grand pour le **Puits** ou le nombre de lignes du **Puits**.

- l'accesseur **`getElements()`** permet de récupérer la **`List`**.
- La dernière méthode d'ajouter dans la classe Tas est **`elementExists(int, int)`** cette méthode doit parcourir la liste des **`Element`** jusqu'à elle trouve un **`Element`** aux coordonnées passés en paramètres. La méthode renvoie **`true`** si un **`Element`** est trouvé, sinon elle renvoie **`false`**.

Modifier la classe **`Puits`**

- Ajouter le(s) attribut(s) requis.
- Ajouter l'accesseur et le mutateur permettent de manipuler les attributs.
- Modifier le(s) constructeur(s) pour qu'après construction le Puits contienne un Tas vide.
- Ajouter un troisième constructeur **`Puits(int,int,int,int)`** où les deux premiers paramètres sont pour la largeur et profondeur du Puits, le troisième paramètre est le nombre d'éléments d'ajouter dans le tas, et le quatrième, le nombre de lignes d'utiliser pour le tas initial.

Valider votre travail avec l'Assignment Centre: 3.6.1

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.6.2 Vérifier le Tas

Travail à faire

Créer, dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`fr.eseo.e3.poo.projet.blox.modele.pieces.TasTest`**

Modifier, dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`fr.eseo.e3.poo.projet.blox.modele.pieces.PuitsTest`**

Implémenter dans ces classes, les tests JUnit5 permettant de tester les fonctionnalités définies pour les classes **`fr.eseo.e3.poo.projet.blox.modele.Tas`** et **`fr.eseo.e3.poo.projet.blox.modele.Pile`**.

- Exécuter vos classes **`TasTest`** et **`PuitsTest`**
- Modifer vos classes de production:
  - **`fr.eseo.e3.poo.projet.blox.modele.Tas`**
  - **`fr.eseo.e3.poo.projet.blox.modele.Puits`**

  et de test:
  - **`fr.eseo.e3.poo.projet.blox.modele.TasTest`**
  - **`fr.eseo.e3.poo.projet.blox.modele.PuitsTest`**

  si necessaire.

Attention, les tests doivent valider les deux cas: Un vecteur de déplacement valide ou invalide.

Valider votre travail avec l'Assignment Centre: 3.6.2

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.6.3 Créer notre propre Exception: Détecter les collisions et les sorties du Puits

Les **`Pieces`** du jeu, défini dans la section 3.2, peuvent se déplacer et tourner dans le **`Puits`** grâce aux interactions de l'utilisateur définies dans la section 3.5. Néanmoins, lorsqu'on regarde les règles du jeu, certaines conditions peuvent rendre les méthodes **`deplacerDe`** ou **`tourner`** inopérantes, bien qu'un vecteur de déplacement valide ait été donné:

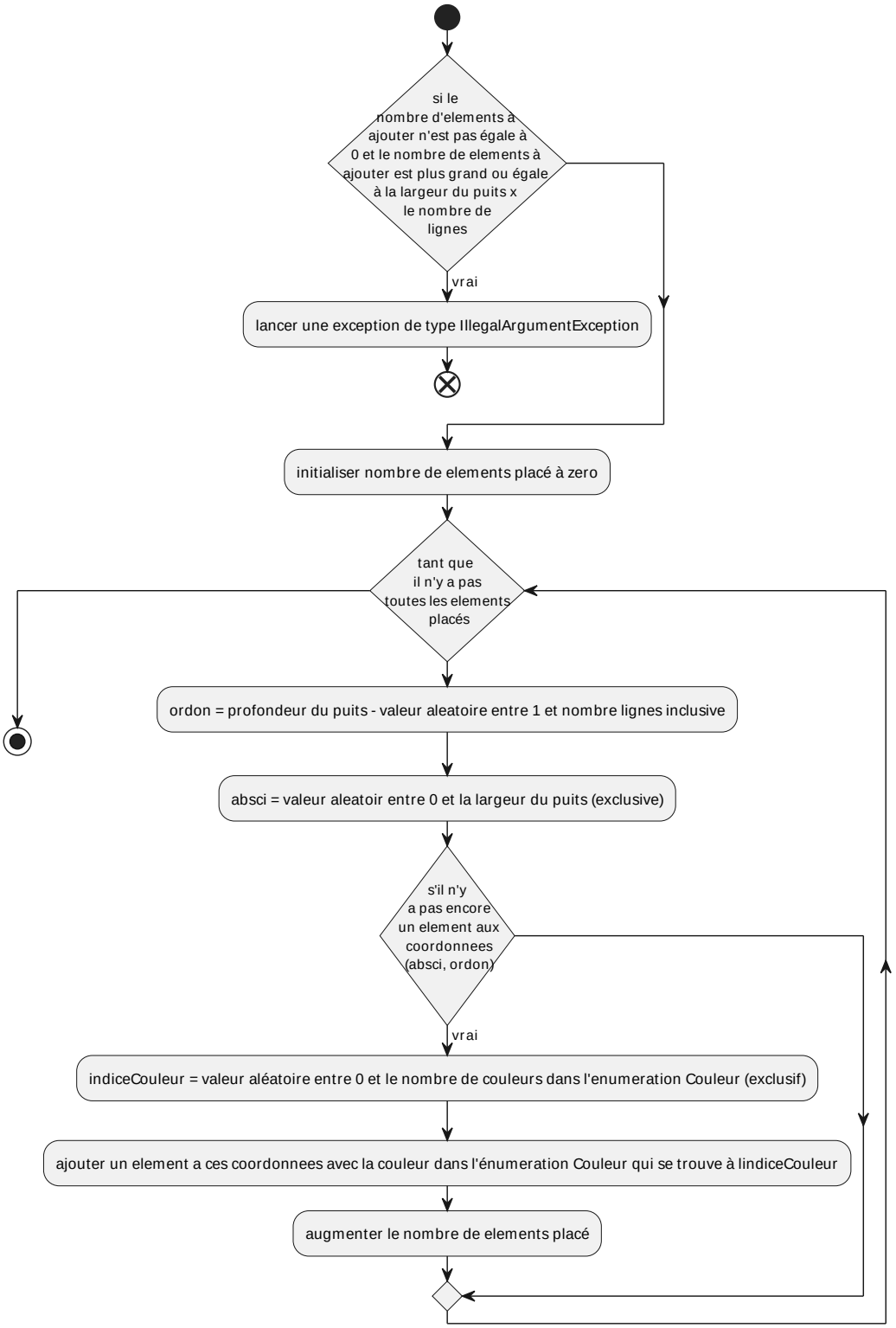


Figure 30: Diagramme d'acitivté pour construireTas

- un des **Elements** de la **Piece** sortirait par la côté gauche ou la côté droite du **Puits**
- un des **Elements** de la **Piece** enterait en collision avec le fond du **Puits** ou un **Element** immobillisé dans le fond du **Puits** (le **Tas**.

Les deux cas cités précédemment représentent de conditions exceptionnelles qui doivent être gérées correctement. Nous avons déjà utilisé les exceptions du type **IllegalArgumentException** pour plusieurs raisons (vecteur de déplacement, taille du puits, ...). De nouveau, nous utiliserons les exceptions pour vérifier les deux conditions ci-dessus. Aucune des classes d'exception fournies par l'API de Java ne correspond précisément à ce cas de figure. Pour cette raison, il sera nécessaire de définir une classe d'exception spécifique à ces cas.

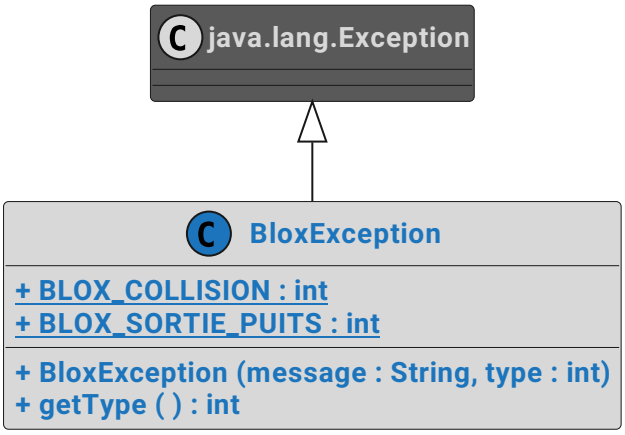


Figure 31: La classe BloxException

La classe **BloxException** du paquetage **fr.eseo.e3.poo.projet.blox.modele** telle qu'elle est représentée dans la figure 31 servira à représenter ces exceptions. La classe fournira un constructeur qui prendra deux paramètres : l'un de type **String** qui spécifiera le message d'erreur associé à l'exception et l'autre de type entier qui permettra de *coder*la raison de la levée d'exception. Ces codes seront des constantes de la classe.

Important!

Étant donné que notre exception sera une sous-classe directe dela classe **java.lang.Exception**, elle sera définie comme une *checked exception*. C'est-à-dire que le compilateur vérifie son utilisation, et s'il n'y a pas les vérifications pour les éventuels levers des exceptions, des erreurs de compilation seront créés.

Travail à faire

Créer la classe **fr.eseo.e3.poo.projet.blox.modele.BloxException**, sous classe de **java.lang.Exception** telle qu'elle est représentée sur la figure 31.

- Définir les deux constantes de classe **BLOX_COLLISION** et **BLOX_SORTIE_PUITS**.
- Écrire le constructeur de la classe. Ce constructeur doit transmettre le message qui lui est passé au constructeur de sa sur-classe et stocker l'entier dans l'instance créée.
- Écrire l'accesseur capable de lire l'entier représentant la cause de l'exception.

Valider votre travail avec l'Assignment Centre: 3.6.3

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.6.4 Refactorisation du code

Nous avons maintenant un **Puits**et une **Piece** -un **Tetromino** - (qui chute lorsque nous tournons la roulette de la souris) et (potentiellement) un **Tas** d'**Element**s au fond du **Puits**. Cependant, le code doit être refactorisé pourqu'il puisse tenir compte des situations exceptionnelles qui peuvent subvenir lors du déplacement ou dela rotation des **Piece**s telles qu'elles sont décrites dans la section 3.6.3. La gestion de ces situations doit interdire aux **Piece**s de sortir du **Puits** (à gauche ou à droite) ou d'entrer en collision avec un **Element** du **Tas** ou le fond du **Puits** - Les **Piece**s peuvent se toucher mais pas de chevaucher. Pour tenir compte de ces situations, la classe **Piece** doit être refactorisée et des conditions doivent être ajoutées à ses méthodes **deplacerDe** et **tourner**.

Travail à faire

Modifier la classe **Tetromino** (et si besoin ses sous-classes) pour qu'elle gère les situations exceptionnelles pouvant subvenir pendant le déplacement:

- Ajouter un variable d'instance de type **Puits** qui permet de stocker le **Puits** qui contient cette **Piece** (avec son accesseur et son mutateur)
- Modifier la méthode **deplacerDe()** de façon à ce que lorsqu'une **Piece** devrait provoquer une collision (avec un **Element** du **Tas** ou avec le fond du **Puits**) si elle effectuait le déplacement qui lui est demandé, les **Element**s qui la composent ne sont pas déplacés et une **BloxException** comportant un message valide et utile est levée. On donnera comme cause à l'exception la valeur **BloxException.BLOX_COLLISION**.
- Modifier la méthode **deplecerDe()** de façon à ce que lorsqu'une **Piece** devrait sortir (à gauche ou à droite) du **Puits** si elle effectuait le déplacement qui lui est demandé, les **Element**s qui la composent ne sont pas déplacés et une **BloxException** comportant un message valide et utile est levée. On donnera comme cause à l'exception la valeur **BloxException.BLOX_SORTIE_PUITS**.
- Modifier la méthode **tourner()** de façon à ce que lorsqu'une **Piece** devrait provoquer une collision (avec un **Element** du **Tas** ou avec le fond du **Puits**) si elle effectuait la rotation qui lui est demandée, les **Element**s qui la composent ne sont pas déplacés et une **BloxException** comportant un message valide et utile est levée. On donnera comme cause à l'exception la valeur **BloxException.BLOX_COLLISION**.
- Modifier la méthode **tourner()** de façon à ce que lorsqu'une **Piece** devrait sortir (à gauche ou à droite) du **Puits** si elle effectuait la rotation qui lui est demandée, les **Element**s qui la composent ne sont pas déplacés et une **BloxException** comportant un message valide et utile est levée. On donnera comme cause à l'exception la valeur **BloxException.BLOX_SORTIE_PUITS**.

Modifier l'interface **Piece** pour que les méthodes **deplacerDe** et **tourner()** sont déclarées de lancer une exception de type **BloxException**.

Avec cette refactorisation, les méthodes **deplacerDe()** et **tourner()** lèvent potentiellement des **BloxException**s, il faut modifier vos classes qui appellent ces méthodes afin qu'elles tiennent compte de ces exceptions.

Travail à faire

Modifier les classes dans les dossiers sources src et test qui font appel à ces méthodes.

- Corriger les erreurs de compilation en ajoutant les blocs de code **try…catch()**.

Valider votre travail avec l'Assignment Centre: 3.6.4

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.6.5 Vérifier la refactorisation

Travail à faire

Modifier, dans le paquetage **`fr.eseo.e3.poo.projet.blox.modele.pieces`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`ElementTest`**
- **`ITetrominoTest`**
- **`OTetrominoTest`**

Implémenter dans ces classes, les tests JUnit5 permettant de tester les fonctionnalités définies pour l'interface **`Piece`**et les clases **`Tetromino`** et **`Element`**.

- Exécuter vos classes **`ElementTest`**, **`ITetrominoTest`** et **`OTetrominoTest`**
- Modifer vos classes de production:
  - **`Element`**
  - **`ITetromino`**
  - **`OTetromino`**

  et de test:
  - **`ElementTest`**
  - **`ITetrominoTest`**
  - **`OTetrominoTest`**

  si necessaire.

Attention, les tests doivent valider que l'exepection de type **`BloxException`** est seulement levée dans les cas exceptionnel cité dans la section 3.6.3

Valider votre travail avec l'Assignment Centre: 3.6.5

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.7 Modifier l'affichage graphique

3.7.1 La VueTas

L'étape suivante dans la création de l'interface de notre application est l'ajout de la classe. **`VueTas`**, la représentation graphique du **`Tas`**, c'est-à-dire des **`Element`**s au fond du **`Puits`**. Les **`Element`**s ont une couleur que nous utiliserons pour les représenter pendant leur chute. Cependant, nous choisirons de modifier légèrement la couleur des **`Element`**s immobilisés dans le **`Tas`**. Comme pour l'élément de référence dans la **`VuePiece`**, nous avons besoin d'implémenter notre propre algorithme. En effet, la méthode **`darker()`** de la classe **`java.awt.Color`** ne sera pas complètement adaptée à notre application. Ainsi, nous allons devoir créer notre propre algorithme pour créer une nouvelle nuance[10]:

1. Définir une constante MULTIPLIER_NUANCE qui sera le facteur de multiplication pour calculer la nouvelle nuance.
2. Récupérer les valeurs entières (entre 0 et 255) qui représentent les différents composants rouge, vert et bleu de la couleur
3. pour chaque composante c:
   - Calculer sa nouvelle valeur

     $$c = c * (1 - MULTIPLIER\_NUANCE)$$

   - Utiliser le transtypage pour créer une valeur entière à partir du résultat obtenu de ce calcul
4. Créer une nouvelle instance de la classe **`java.awt.Color`** en utilisant les nouvelles valeurs pour les composants rouge, vert et bleu.

[10] En théorie de couleur, une nuance est le mélange d'une couleur avec du noir, ce qui réduit la légèreté

L'objet **Tas** n'est jamais modifié (aucun **Tas** n'est jamais créé pendant le jeu, seul le contenu du tableau lui correspondant est modifié). Ainsi, pour afficher l'état du **Tas**, une **VueTas** se contentera d'afficher le contenu de ce tableau. La classe **VueTas** a donc besoin de connaître son **Tas**.

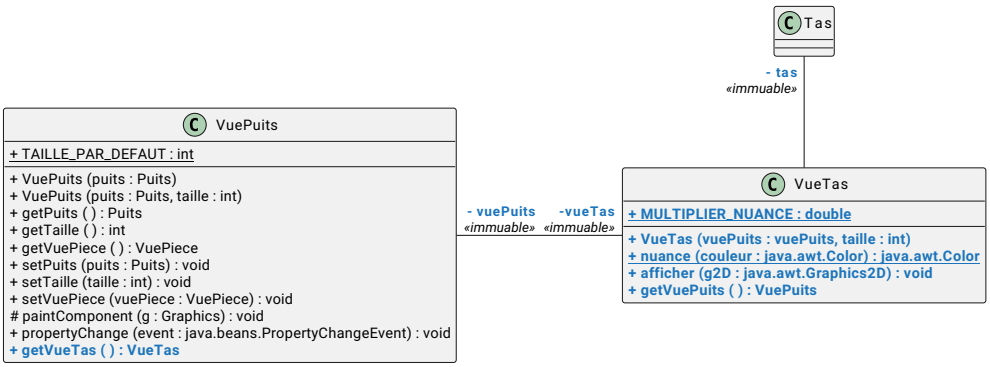


Figure 32: La classe VueTas

Travail à faire

Créer la classe **fr.eseo.e3.poo.projet.blox.vue.VueTas** telle qu'elle est représentée dans la figure 32.

- Définir la constante de classe **MULTIPLIER_NUANCE** avec une valeur réelle[a]. m:

$$m \in \mathbb{R}, m \in]0.0, 1.0[$$

- Le constructeur doit prendre un **VuePuits** en argument et créer la relation entre la **Tas** et sa **VueTas** (le **Tas** est récupéré depuis le **Puits** associé au **VuePuits**
- La méthode *de classe* nuance() prend en paramètre une instance de la classe **java.awt.Color** et retourne une nouvelle instance de cette classe en utilisant l'algorithme ci-dessus pour sa création.
- dans la méthode **afficher** en utilisant l'instance de **java.awt.Graphics2D** fournie en paramètre, dessine en utilisant ses méthodes **setColor()**, **nuance** et **fill3DRect()** les différents **Element**s du **Tas** en assurant que l'effet 3D semble être en dessous de la surface.
- Écrire les accesseurs et mutateurs.
  Modifier la classe **VuePuits** pour que la relation entre **VuePuits** et **VueTas** soit implémentée dans le constructeur de **VuePuits** et que la méthode **paintComponent** appelle (à l'endroit approprié) la méthode d'affichage de **VueTas**, pour dessiner le **Tas**. Par ailleurs, les accesseurs et mutateurs pour la **VueTas**.

[a] Le choix de cette valeur est libre, et dépend de vos propres goûts

Valider votre travail avec l'Assignment Centre: 3.7.1

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.7.2 Vérifier la VueTas

Travail à faire

Modifier (si besoin), dans le paquetage **fr.eseo.e3.poo.projet.blox.vue** qui se trouve dans le dossier source test, la classe suivante:

- **fr.eseo.e3.poo.projet.blox.vue.VuePuitsAffichageTest**

Lancer l'application de test et vérifier manuellement qu'un **Tas** est bien représenté dans la fenêtre d'affichage.

- Modifer votre classe de production
  `fr.eseo.e3.poo.projet.blox.vue.VuePuits`
  et de test
  `fr.eseo.e3.poo.projet.blox.vue.VuePuitsAffichageTest`
  si necessaire.

Valider votre travail avec l'Assignment Centre: 3.7.2

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

Attention, ne touchez pas le clavier ni la souris pendant l'exécution de tests par l'Assignment Centre. Lors des tests de l'interface graphique, l'Assignment Centre affiche des fenêtres sur l'écran et peut interagir avec. Si vous manipulez le clavier ou la souris pendant ces tests, l'Assignment Centre peut enregistrer des erreurs.

3.8 Finir la version basique du jeu

3.8.1 Modélisation de la gravité

Un aspect très important du jeu Falling Blox est le fait que les **`Piece`**s sont soumises à la gravité et chutent dans le **`Puits`**. Dans cette version du jeu, la force de gravité -et donc la vitesse de chute des **`Piece`**s- sont supposées constantes. La gravité peut donc être exprimée en carreaux/unité de temps[11]. Une **`Piece`** chute tatnt qu'elle n'a pas atteint le fond du **`Puits`** ou tant qu'un **`Element`** de la **`Pile`** ne se trouve pas directement sous elle. Dans ces deux cas, la **`Piece`** se *disloque* et ses **`Element`**s sont ajoutés au **`Tas`**. Pour implementer ceci, il est necessaire de modifier les classes de **`Puits`** et **`Tas`**. La classe **`Puits`** devra définir la méthode **`gravite()`**, qui appellera la méthode **`deplaceDe()`** de **`Piece`** pour déplacer la pièce actuelle d'un carreau vers le bas du **`Puits`**. Cette méthode devra gérer convenablement les exceptions qui pourraient être levées par la méthode **`deplacerDe`**. En particulier, lorsque le **`Piece`** arrive au fond du **`Tas`** elle doit appeler la méthode **`gererCollision()`** chargée de gérer la collision: c'est-à-dire appeler la méthode **`ajouterElements()`** de la classe **`Tas`** et créer une nouvelle pieceSuivante à l'aide de la classe **`UsineDePiece`** en mettant à jour la pieceActuelle.

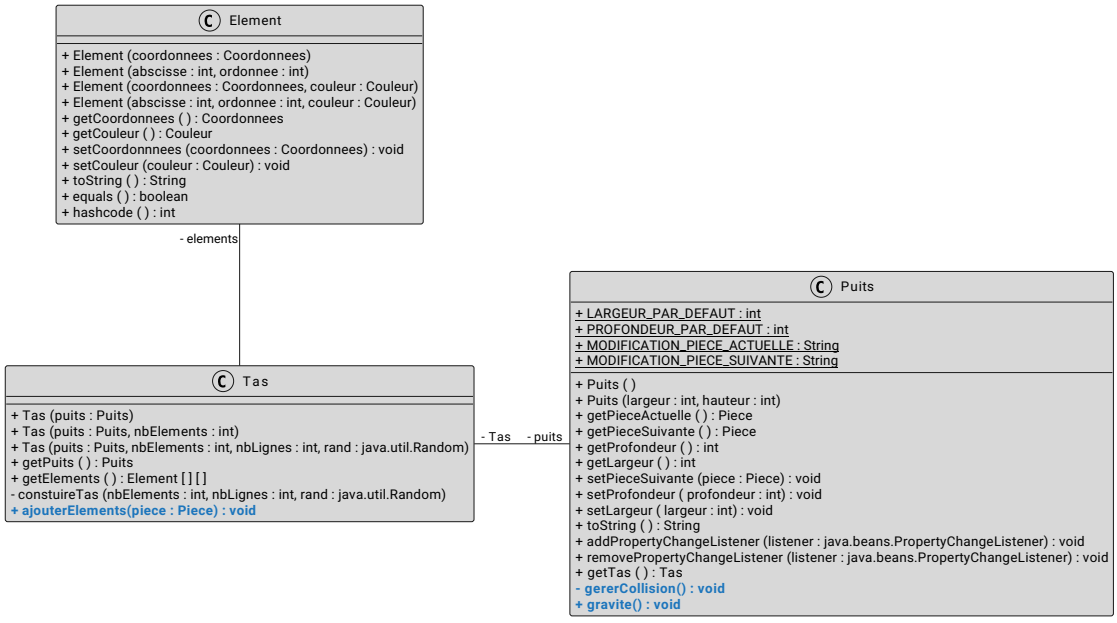


Figure 33: Modélisation de la gravité

[11] Ce sera seulement dans la section 3.8.3 quand nous automatisons la gravité que l'unité de temps nous intéressera

La méthode **addElements()** de la classe **Tas**, est chargée d'ajouter au **Tas** les **Element**s de la **Piece** passée en paramètre.

Travail à faire

En se basant sur le diagramme de classes dans la figure 33…
Modifier la classe **Tas** comme suit:

- Créer la méthode **ajouterElements()**, qui ajoute individuellement tous les **Element**s de la **Piece**, donnée en paramètre, dans le tableau à deux dimensions du **Tas**.

Modifier la classe **Puits** comme suit:

- Créer la méthode privée **gererCollision()** qui appelle la méthode **ajouterElements()** du **Tas** en lui passant en paramètre la pieceActuelle. Une fois les **Element**s ajoutés au **Tas**, la méthode **gereCollision()** doit la remplacer pas la pieceSuivante et créer aléatoirement une nouvelle pieceSuivante à l'aide de la classe **UsineDePiece**.
- Créer la méthode **gravite()**, qui tente de déplacer la pieceActuelle dans le **Puits** d'une case vers le bas. Cette méthode doit détecter et gérer correctement les situations exceptionnelles qui peuvent subvenir.
- C'est possible que vous deviez modifier la méthode **deplacerDe()** des classes qui implémente **Piece** pour qu'elles génèrent la bonne exception pour quand la **Piece** arrive en bas ou fait une collision avec le **Tas**.

Valider votre travail avec l'Assignment Centre: 3.8.1
L'Assignment Centre vérifie que la structure de votre code est correcte.

3.8.2 Vérifier la gravité

Travail à faire

Modifier, dans le paquetage **fr.eseo.e3.poo.projet.blox.modele** qui se trouve dans le dossier source test, les cas de test JUnit 5 suivants:

- **fr.eseo.e3.poo.projet.blox.modele.pieces.TasTest**
- **fr.eseo.e3.poo.projet.blox.modele.pieces.PuitsTest**

Implémenter dans ces classes, les tests JUnit5 permettant de tester les fonctionnalités définies pour la classe **fr.eseo.e3.poo.projet.blox.modele.Tas** et pour la classe **fr.eseo.e3.poo.projet.blox.modele.Puits**.

- Exécuter vos classes **TasTest** et **PuitsTest**
- Modifer vos classes de production:
  - **fr.eseo.e3.poo.projet.blox.modele.Tas**
  - **fr.eseo.e3.poo.projet.blox.modele.Puits**

  et de test:
  - **fr.eseo.e3.poo.projet.blox.modele.TasTest**
  - **fr.eseo.e3.poo.projet.blox.modele.PuitsTest**

  si necessaire.

Attention, les tests doivent valider les deux cas: Un vecteur de déplacement valide ou invalide.

Valider votre travail avec l'Assignment Centre: 3.8.2
L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.8.3 Automatiser la gravité

Pour l'instant, nos pièces ne chutent que lorsque nous le décidons et que nous manipulons la roulette de la souris. Ce n'est évidemment pas un comportement acceptable pour un jeu de Tetris. Nous devons trouver

un moyen de faire tomber automatiquement les pièces et faire en sorte que la méthode **`gravite()`** qui implémente la gravité puisse être appelée automatiquement à intervalles réguliers. Il existe plusieurs façons en Java d'implémenter des événements récurrents; en particulier s'ils sont périodiques. Les solutions possibles sont:

- Des **`Thread`**s séparés
- Des boucles que nous occupons inutilement pour qu'un tour de boucle s'exécute en un temps déterminé
- Les Timers (les compteurs)

La première solution, bien que la plus adaptée à notre problème, est aussi la plus difficile à mettre en œuvre. La deuxième semble être une solution simple, mais elle est loin[12] d'être efficace. En effet, pendant qu'une boucle est occupée à ne rien faire pour perdre du temps, elle ne peut rien faire d'autre (que ne rien faire…). La troisième solution avec les Timers sera la solution que nous choisirons.

Java nous offre deux types de compteur: **`java.util.Timer`** et **`javax.swing.Timer`**. Il existe des avantages et des inconvénients aux deux types de compteur. Pour notre jeu, nous voulons trouver un compromis entre une rapidité / précision et facilité d'utilisation. Il n'est pas très important dans notre cas d'avoir une précision à l'échelle des millisecondes, nous choisirons le **`javax.swing.Timer`** parce que toutes les x millisecondes, une instance de cette classe génère un **`ActionEvent`** - c'est comme toutes les x millisecondes, quelqu'un effectue un clic sur un bouton. Il ne nous reste donc qu'à écrire un gestionnaire d'événement de type **`ActionListener`** pour gérer cet événement, grâce à sa méthode **`actionPerformed`** à chaque fois qu'il se produit.

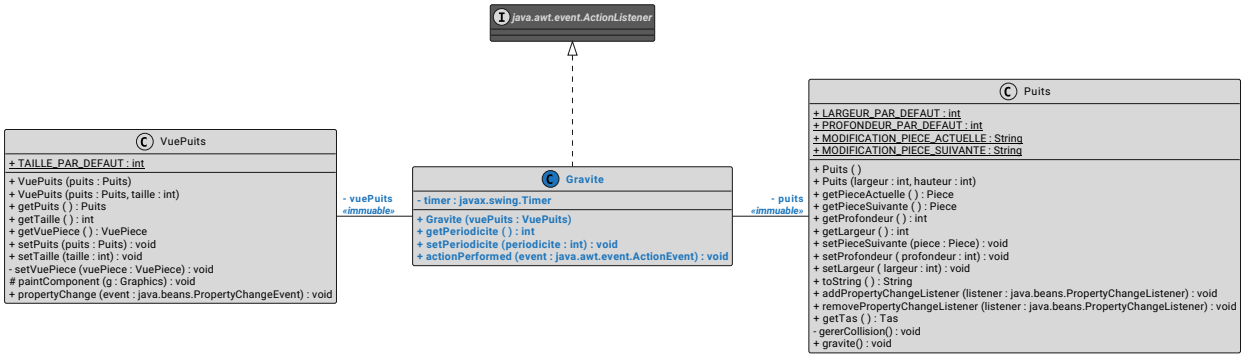


Figure 34: La classe Gravite

Travail à faire

Créer la classe **`fr.eseo.e3.poo.projet.blox.controleur.Gravite`** comme définie dans la figure 34. Cette classe dvra implémenter l'interface **`java.awt.event.ActionListener`**.

- Le constructeur doit créer le lien entre **`Gravite`**, **`VuePuits`** et **`Puits`** et fixera la périodicité du timer en millisecondes.
- Dans le constructeur, créer et démarrer un swing timer qui créera un événement périodique - l'instance lui-même de **`Gravite`** doit être enregistré comme un listener de ce timer.
- La méthode **`actionPerformed()`** qui sera appelée après chaque intervalle périodique; appellera la méthode **`gravite()`** de l'instance de **`Puits`** concernée et qui se chargera de raffichir la **`VuePuits`**.
- *ajouter l'accesseur et le mutateur.

Vérifier bien que dans la constructeur de VuePuits, vous ne demarrez pas la gravité.

[12] très loin!

Valider votre travail avec l'Assignment Centre: 3.8.3

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.8.4 Vérifier la gravité automatique

Travail à faire

Créer, dans le paquetage **`fr.eseo.e3.poo.projet.blox.controleur`** qui se trouve dans le dossier source test, le cas de test JUnit 5 suivant:

- **`fr.eseo.e3.poo.projet.blox.modele.pieces.GraviteTest`**

Implémenter dans cette classe, une application qui crée une fenêtre avec une représentation visuelle d'un **`Puits`**. Faire en sorte qu'une **`Piece`** est bien ajouté quelque part dans le **`Puits`** et que sa visualisation graphique (une **`VuePuits`**) est visible. Vérifier que la pieceActuelle descend le **`Puits`** grâce au timer. Vous devez ajouter le code pour créer une instance de la classe Gravite et pour demarrer le timer dansla classe Gravite dans le code de votre classe de test.

Valider votre travail avec l'Assignment Centre: 3.8.4

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

3.8.5 Afficher un PanneauInformation avec la visualisation de la piece suivante

Nous allons créer un autre sous-classe de JPanel qui permet d'afficher la piece suivante.

Travail à faire

Créer la classe **`fr.eseo.e3.poo.projet.blox.vue.PanneauInformation`** comme suit:

- C'est une sous-classe de **`javax.swing.JPanel`** aui implements l'interface **`java.bean.PropertyChangeListener`**.
- Son constructeur prendre en paramètre une instance de la classe **`Puits`**. Le constructeur doit enregistrer l'instance en tant que **`PropertyChangeListener`** pour le **`Puits`** passé en paramètre. Le constructeur doit régler la taille de préférence pour être 70 x 70 pixels.
- Dans la méthode **`propertyChanged`**, vérifier si le changement est un changement de la prochaine pièce, et si c'est le cas de créer une nouvelle VuePiece avec comme Piece, la nouvelle valeur (de le **`PropertyChangeEvent`** et comme taille la valeur 10. Stocker cette VuePiece dans une variable d'instance.
- redéfinir la méthode **`paintComponent()`** pour que si la variable d'instance contienne une VuePiece, qu'elle appel la méthode **`afficherPiece()`** de la classe **`VuePiece`** pour l'afficher.

Valider votre travail avec l'Assignment Centre: 3.8.5

L'Assignment Centre vérifie que la structure de votre code est correcte.

3.8.6 Tester le jeu

Travail à faire

Créer la classe principale **`FallingBloxVersion1`**.
En exécutant cette classe, l'utilisateur doit être capable de jouer notre jeu. L'utilisateur doit avoir la possibilité de démarrer le jeu dans trois façons:

- Sans arguments de la ligne de commande
  - Le jeu commencent normalement sans **`Elment`** dans le **`Puits`**
- Avec qu'un seul argument en ligne de commande:
  - Une valeur entière qui défine combien d'**`Element`**s doit être ajouté dans le **`Tas`** du **`Puits`** au début de jeu.
- Avec deux arguments en ligne de commande:
  - La première valeur entière qui définie combien d'**`Element`**s doit être ajouté dans le **`Tas`** du **`Puits`** au début de jeu, et la deuxième qui définie combien des lignes doivent être utilisés

Le titre de la fenêtre doit être Falling Blox, et que:

- elle contient au centre un **`VuePuits`**
- elle contient à droite (**`BorderLayout.EAST`** un **`PanneauInformation`**
- la taille de la fenêtre ne peut pas être modifié par l'utilisatuer
- la fenêtre est positionné au centre de l'écran
- que une piece commence de tomber

Valider votre travail avec l'Assignment Centre: 3.8.6

L'Assignment Centre vérifie la fonctionnalité de votre classe est correcte. Les tests de l'Assignment Centre ne doivent en aucun cas être utilisés pour remplacer vos propres tests.

4 Les Extensions

Nous avons créé une application qui fonctionne, avec une implémentation basique qui nous permet de jouer au jeu de FallingBlox, par contre dans son état actuel, il y a pas mal de fonctionnalités qui manquent. C'est à vous de jouer, ajouter les fonctionnalités de votre jeu. Les seules règles que vous devez suivre, sont:

1. de garder les bases comme telles, et qu'après n'importe quel modification, l'Assignment Centre valide toujours votre travail - pas de régression!
2. des que c'est possible, d'utiliser la fonctionnalité déjà en place, par exemple reflechir comment le **`PropertyChangeSupport`** / **`PropertyChangeListener`**s peuvent être utilisé pour ajouter plus de fonctionnalité.

Si vous en avez besoin, vous pouvez:

- créer des nouvelles classes principales (**`FallingBloxVersionX`**) pour exécuter votre application avec les extensions
- ajouter les méthodes et les attributs en plus dans les classes
- créer les sous-classes des classes déjà défini
- ...

4.1 Des idées pour les extensions

Avant d'ajouter n'importe lequel extension, nous vous demandons en premier d'implémenter les deux extensions suivantes:

- 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).
- 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)

Ensuite, libre choix de faire avancer la fonctionnalité de FallingBlox. À vous de choisir la fonctionnalité d'ajouter, si vous êtes courte des idées, la liste ci-dessous est une liste (non exhaustive) des idées que vous pouvez implémenter pour créer une application FallingBlox complète.

- Ajouter le score (par exemple 10 points pour chaque ligne complète dans le tas)
- Permettre le joueur de faire un échange entre la pièce actuelle et la pièce suivante
- Modifier la vitesse de la gravité (chaque fois que le joueur a complété 10 (15, 20 ...) lignes, augmenter la vitesse)
- Créer les pentominos[13] - en se basant sur les principes utilisés pour les tetrominos.
  - Par exemple, chaque fois que la vitesse augmente, générer une Pentomino
  - La création de ces pentominos doivent être géré par l'**`UsineDePiece`** - Mais ne peut pas modifier ni la fonctionnalité ni l'ordre de génération des Tetrominos.
    - Par exemple créer la méthode **`genererPentomino()`**

[13] voir https://tetris.wiki/Pentomino pour les différents pentominos

- Ajouter la descente directe, quand l'utilisateur clic sur la roulette de la souris, la pièce se trouve en bas du puits.
- Utiliser les boutons physiques pour gérer le mouvement et la rotation des pièces
- Utiliser le clavier pour la rotation et le mouvement des pièces
- Ajouter un *image fantôme* pour visualiser où la pièce va trouver quand il est tombé tout en bas.
- Gérer la fin de la partie. Pour l'instant, il n'y a pas les vérifications pour un Puits qui est remplie.
- Ajouter un tableau des meilleurs scores qui est enregistré entre les exécutions du jeu
- Ajouter un bouton qui permettre de mettre en pause une partie du jeu
- Ajouter un mode de jeu à deux joueurs, soit:
  - Deux joueurs sur la même machine
  - Deux joueurs sur les machines différents connectés via le réseau
  - Un joueur contre l'ordinateur

  Exemples de mode de jeu:
  - Coopératif (les deux joueurs contrôle deux pièces qui tombent dans le même puits).
  - Un contre un
    * Deux puits dans l'interface (un pour chaque joueur[14]), quand un des joueurs construit une ligne complète, le tas de l'autre joueur est monté par une ligne qui est remplie aléatoirement par quelques éléments.
    * Un puits, plus haut, avec le tas au milieu, un joueur contrôle une pièce qui tombe d'en haut, l'autre une pièce qui remonte d'en bas. Chaque fois qu'un joueur créé une ligne complète, le tas bouge vers l'autre joueur, le jeu se termine quand un des deux joueurs ne plus pas avoir une pièce.

[14] Dans le jeu via réseau, le deuxième affiche l'état de jeu, mais n'est pas interactif