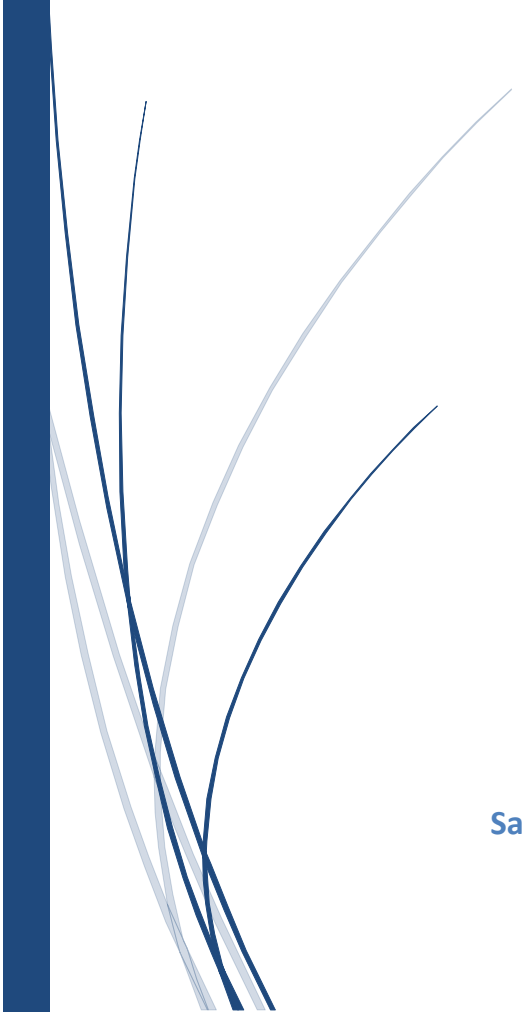
29/12/2019

Rapport Phineloops

Par SFB



Sania Ur Rehman – Fatima Zahra Marhous – Boris Martin

Table des matières

Introduction 3

I. Architecture 4

1. Pattern MVC 4

2. Diagramme de classe 5

II. Les fonctionnalités apportées 12

1. Générateur de niveaux 12

2. Vérificateur de solution 13

3. Solveur de niveaux 14

a) Choco solver 14

b) Stack solver 16

4. Visualisation de niveaux 17

III. Déroulement et répartition des tâches 18

IV. Performances des solvers 19

Conclusion 21

Annexes 22

Introduction

Dans le cadre de l'UE Java avancé du Master 1 MIAGE, nous avons eu à développer un jeu, infinity loop. Pour cela, nous avons programmé plusieurs fonctionnalités du jeu : générer un niveau, vérifier la solution, résoudre un niveau et créer une interface graphique pour l'utilisateur.

L'objectif principal de ce projet est de créer le jeu tout en fournissant une architecture d'application qui respecte les principes de base du Java Objet.

Nous allons tout d'abord monter l'architecture de notre application avec un diagramme UML puis présenter les fonctionnalités de notre application et les choix de conceptions. Nous allons ensuite exposer les performances de notre logiciel et pour finir expliquer le déroulementde notre programmation en équipe.

I. Architecture

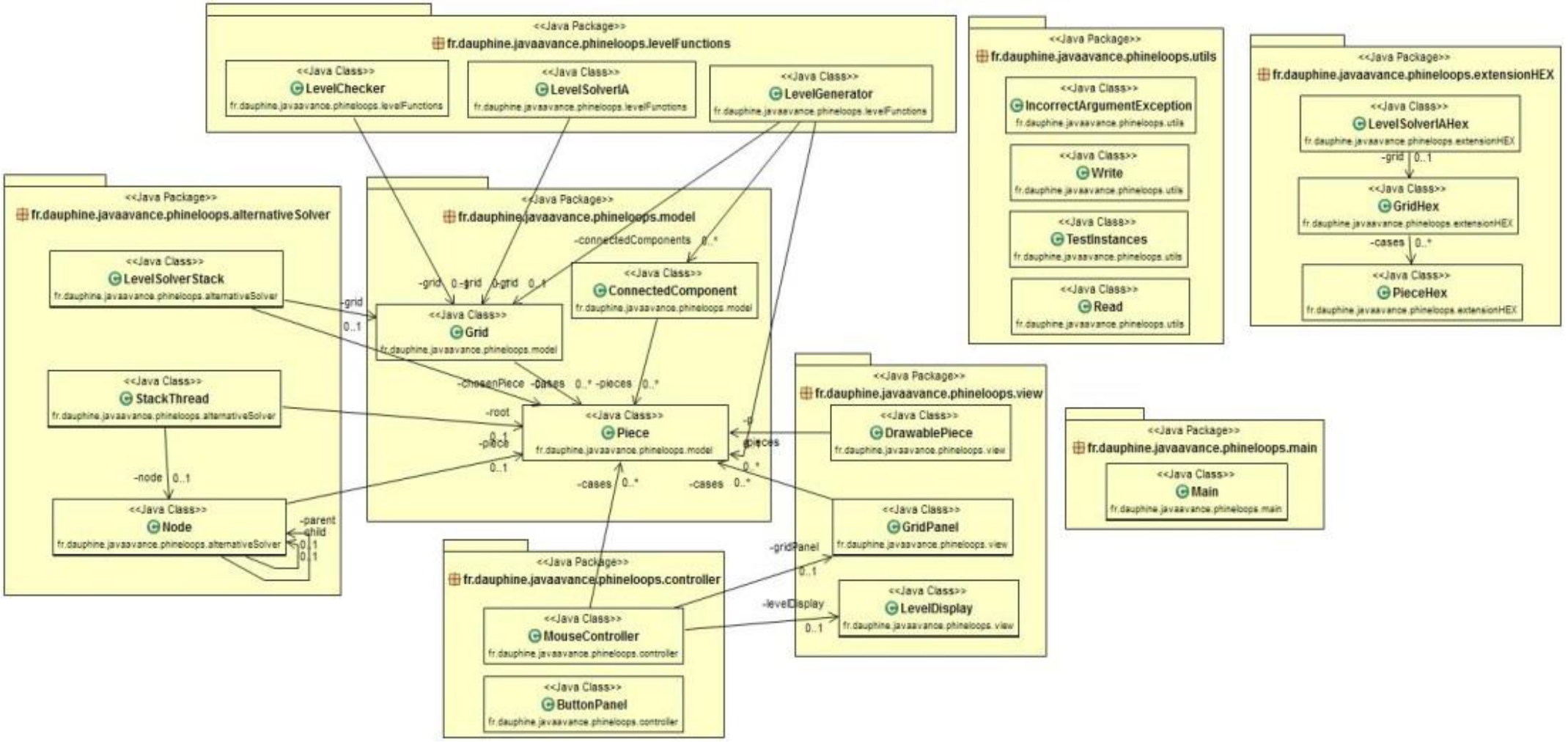
1. Pattern MVC

Dans l'architecture de notre projet, nous avons choisi d'appliquer le pattern MVC vu en cours. Nous avons ainsi un package Model, les objets de base qui structure un niveau. C'est sur ces objets que nous allons effectuer notre résolution. Ce package comprend ainsi les classes Piece, Grid et ConnectedComponent.

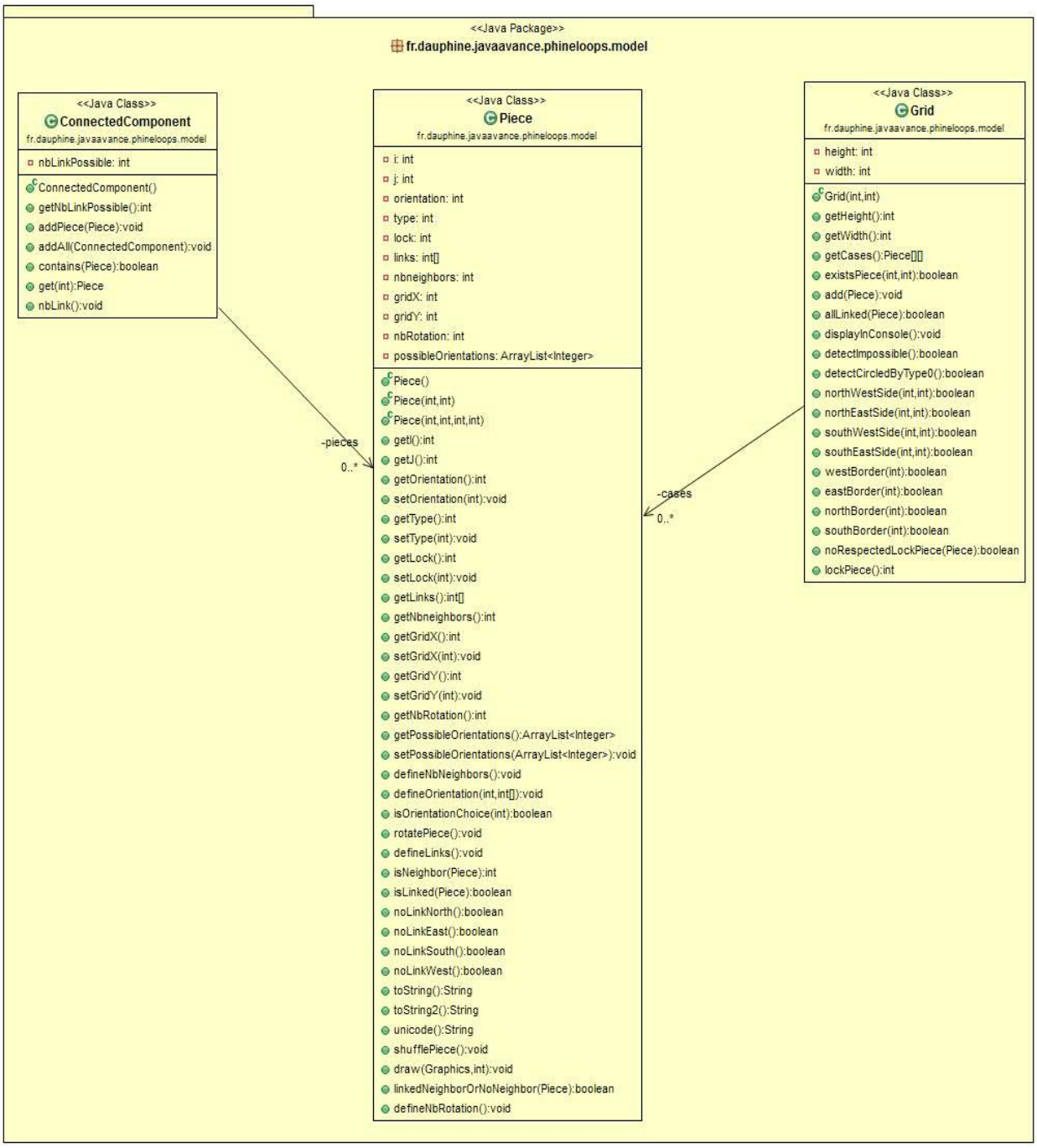
Dans le package View, nous avons les différentes classes permettant la représentation visuelle d'un niveau. Ainsi pour une pièce, un objet DrawablePiece associé est créé, de même pour une grille avec GridPanel et enfin LevelDisplay permettant de représenter notre niveau.

Enfin dans le package Controller, nous avons donc les classes permettant de faire le lien entre les packages Model et View. Il permet ainsi via des boutons d'appliquer les différentes actions à notre niveau (check, suffle, solve et new) et aussi de permettre à l'utilisateur de faire bouger les différentes pièces.

2. Diagramme de classe



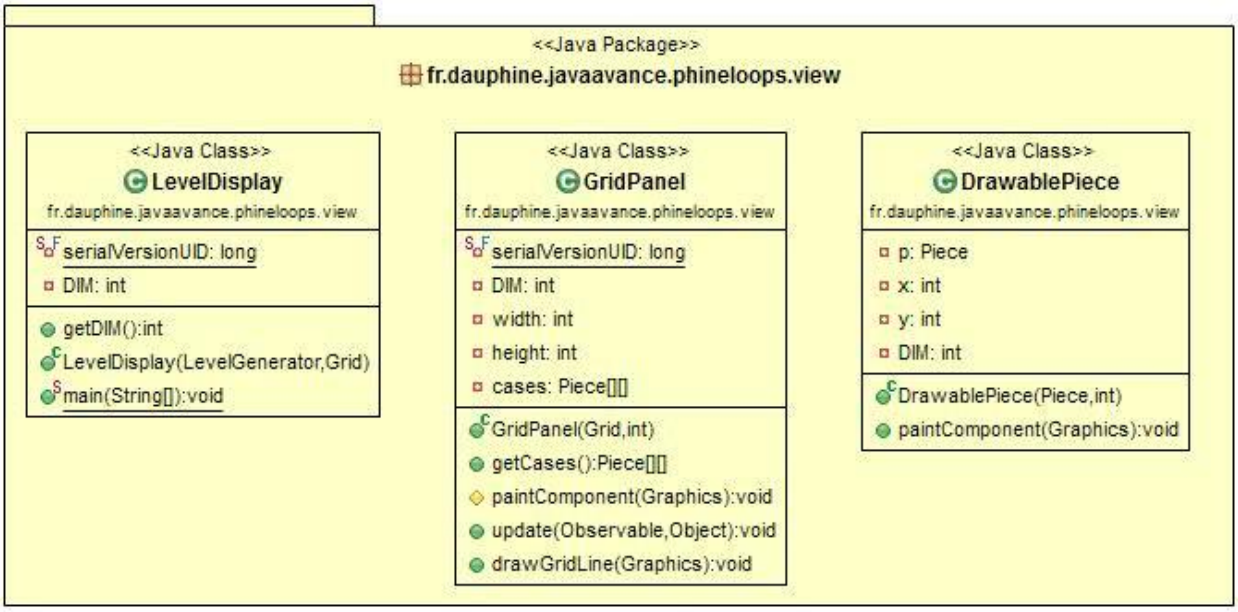
- Package model



Le package model contient les 3classes basique du jeu :

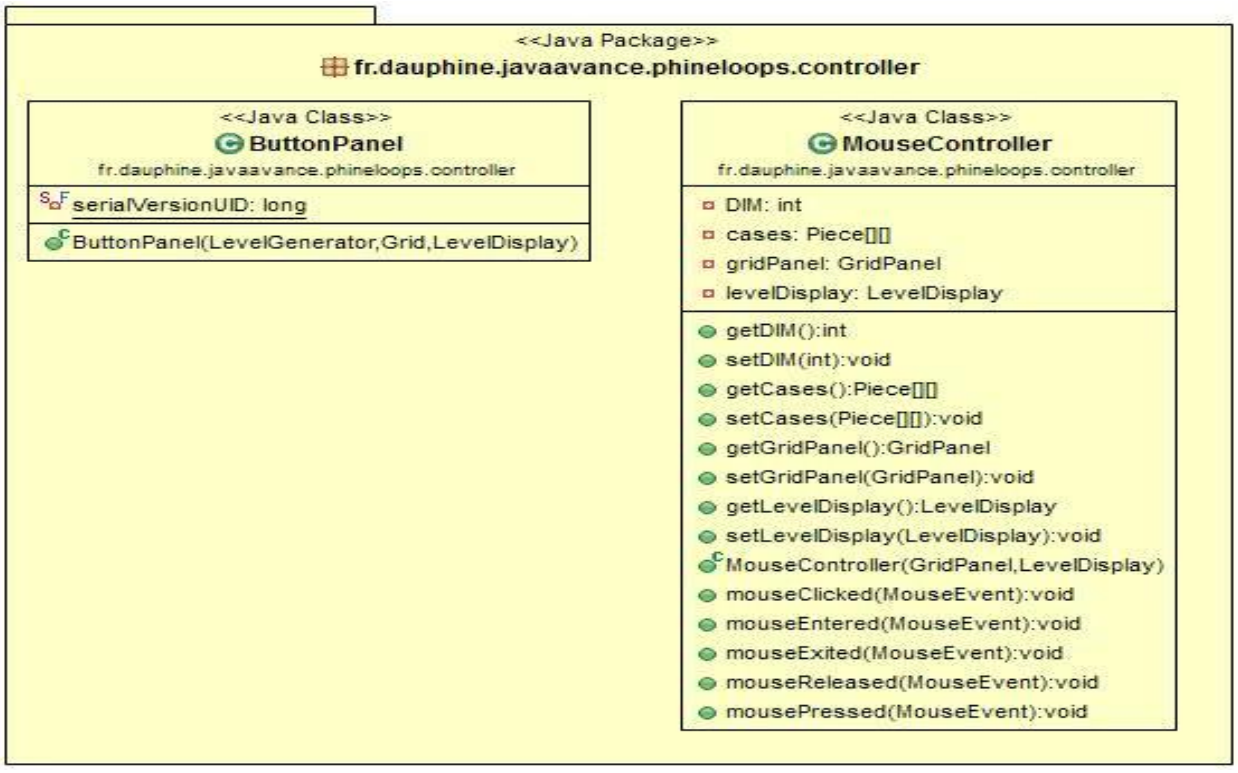
- La classe Piece : représente les différentes pièces qu'on peut trouver dans la grille du jeu.
- La classe Grid : représente la grille du jeu selon les dimensions choisi par l'utilisateur.
- La classe ConnectedComponent : définie une composante connexe qu'on peut trouver entre les pièces après la résolution d'une grille.

- Package view



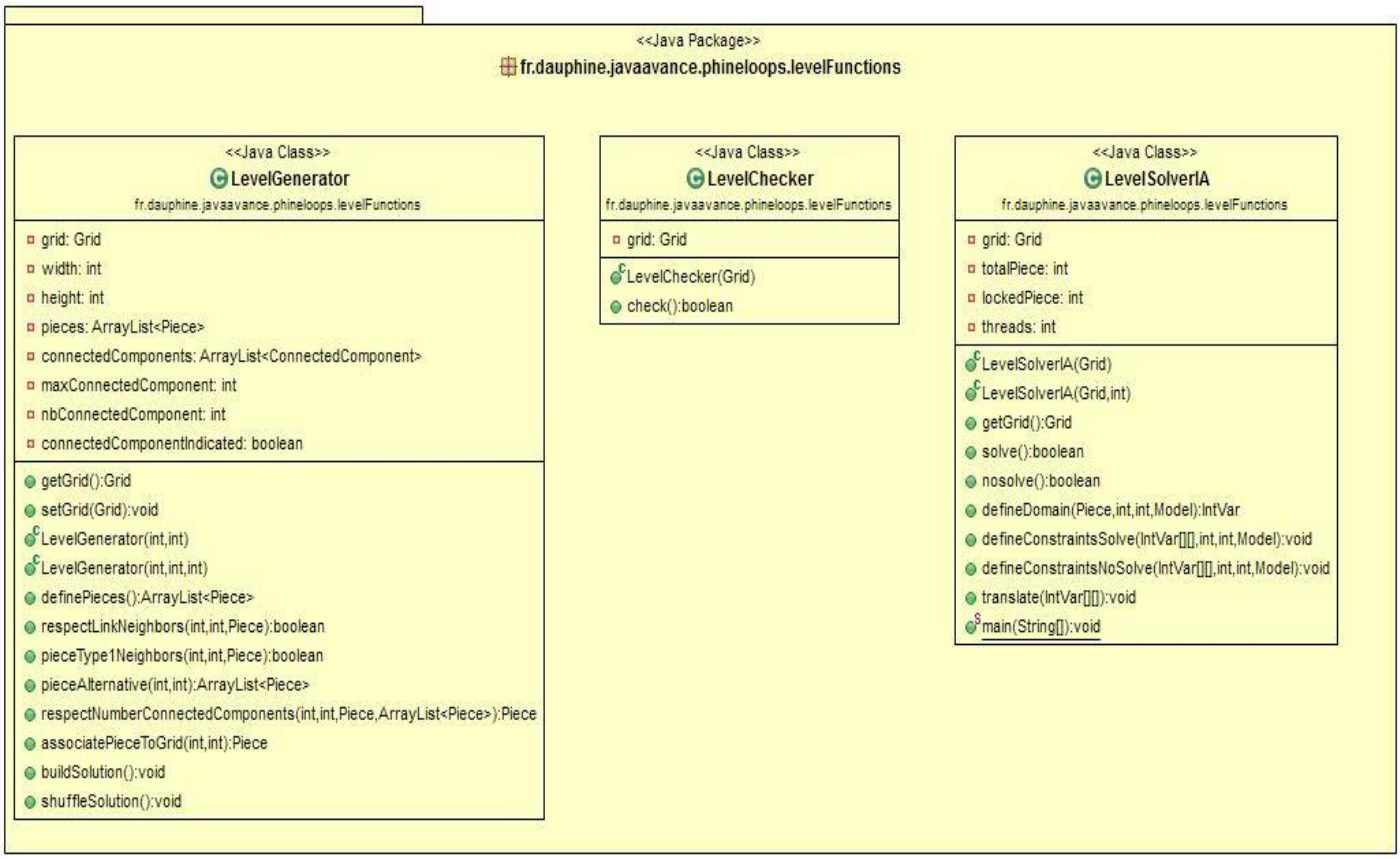
Le package view permet l'affichage du package model avec les classes : DrawablePiece, GridPanel et LevelDisplay.

- Package controller



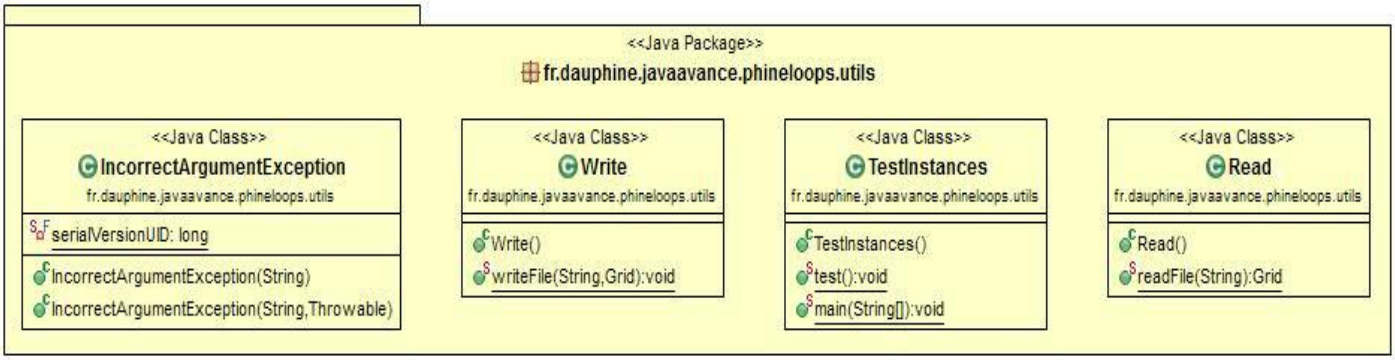
Le package controller met à jour l'interface graphique.

- Package levelFunctions



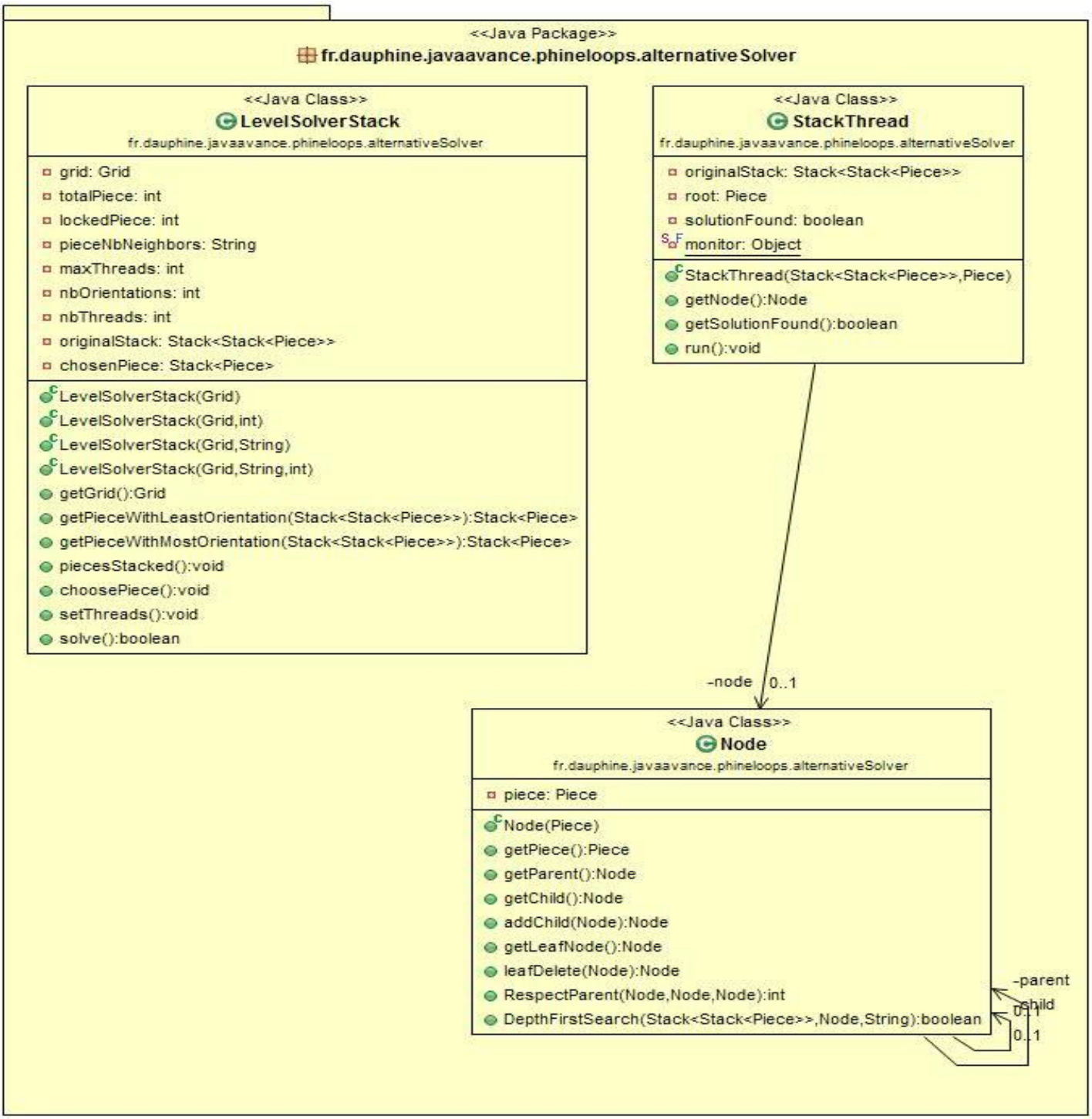
Le package LevelFunction contient les fonctionnalités principales, apportées au jeu :

- La classe LevelGenerator : permet de générer un niveau du jeu.
- La classe LevelChecker : vérifie si une solution d'une grille est bonne ou pas.
- La classe LevelSolverIA : permet de résoudre une grille donnée, en utilisant l'algorithme CSP vu en cours de l'Intelligence artificiel avec la bibliothèque de chocosolver.
- Package utils



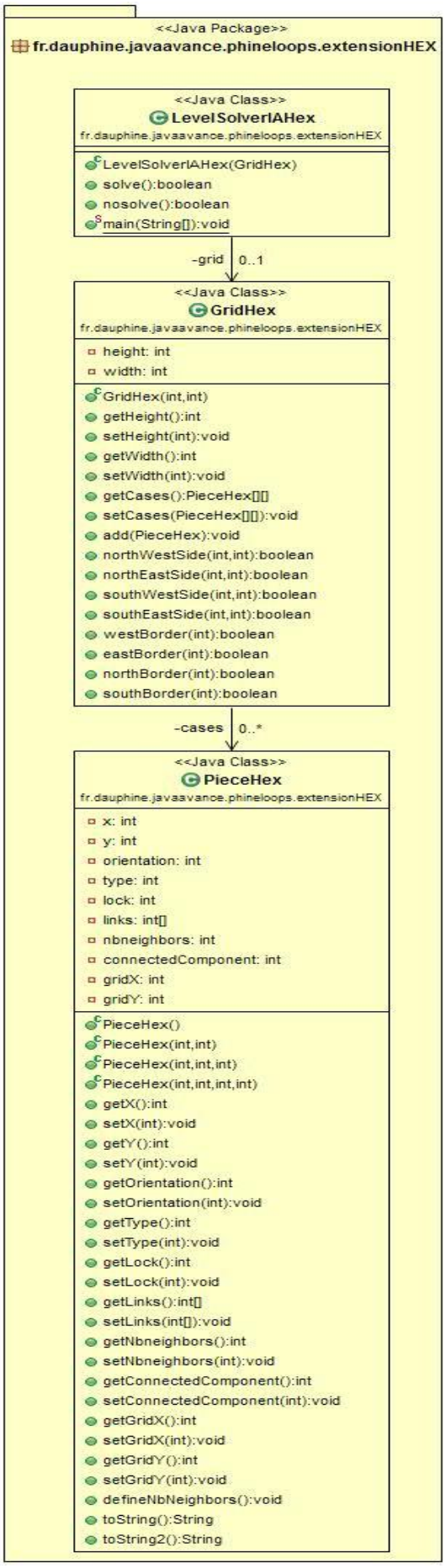
Le package utils, comme son nom l'indique, contient les classes qui permet le bon fonctionnement du programme.

- Package alternativeSolver :



Le package alternativeSolver contient un autre solveur basé sur le sujet du projet.

- Package extensionHEX



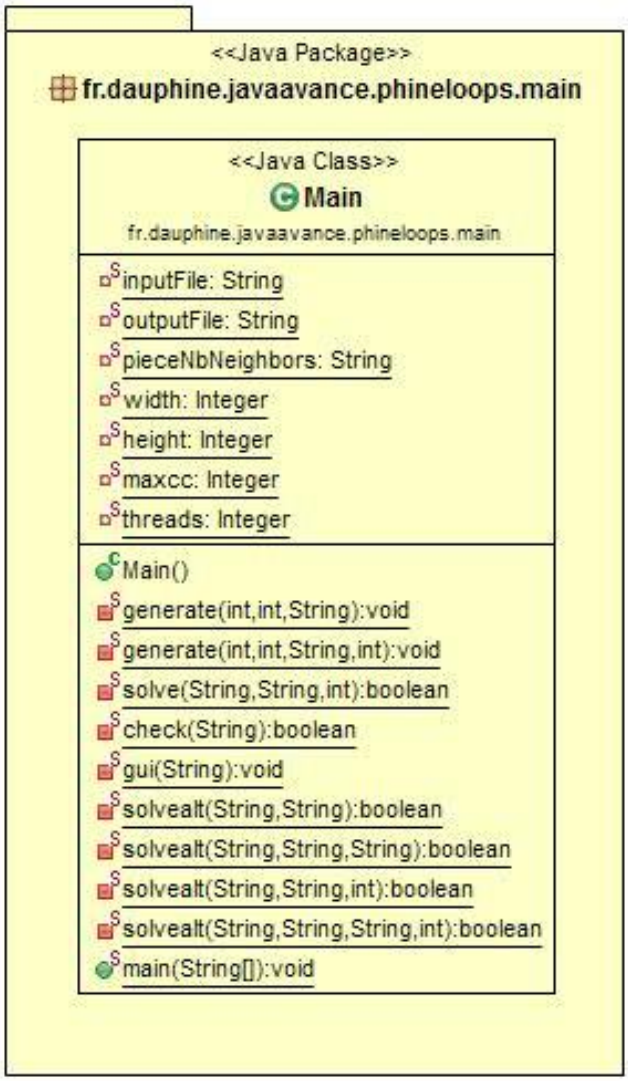
Le package extensionHex contient les classes qui permettent la généralisation de la version HEX du jeu.

- La classe PieceHex : modélise les pièces de la version HEX du jeu, où on a 14 types de pièces avec jusqu'à 6 orientations.

- La classe GridHex : représente la grille de la version HEX du jeu.

La classe LevelSolverIAHex : permet de résoudre une grille HEX avec la méthode solve() du choco-solver.

- Packege main :



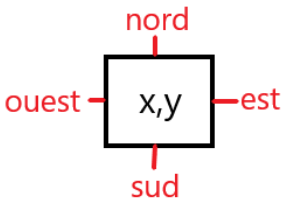
Le package main contient la classe main du projet qui permet de lancer les différentes fonctionnalités à partir du terminal.

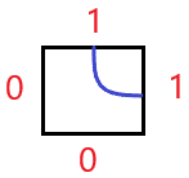
II. Les fonctionnalités apportées

Notre programme dispose de plusieurs fonctionnalités : il peut générer des niveaux à résoudre pour l'utilisateur, vérifier que la résolution du niveau est belle et bien vraie ainsi que de résoudre lui-même un niveau. Pour pouvoir jouer, nous proposons une interface graphique au joueur.

1. Générateur de niveaux

Dans notre programme un niveau est représenté par une grille (tableau à 2 dimensions). Chaque case de la grille contient une pièce. Chacune des pièces a quatre bords : nord, est, sud et ouest. Une pièce peut se lier à une ou plusieurs de ses voisines via ses bords. Si une pièce possède un lien vers un bord, ce bord prend la valeur 1 sinon 0. Une solution réalisable d'un niveau a toutes les pièces de la grille liées entre elles.





L'utilisateur peut générer un niveau en spécifiant les dimensions de son niveau : le nombre de pièces à l'horizontal et à la verticale. Pour construire un niveau, nous partons de la première case de la grille (0.0) et choisissons la pièce qui respecte les cases voisines à son nord et ouest. Par respect, nous voulons dire que le bord commun avec sa voisine (nord ou ouest) ait la même valeur pour les deux pièces. De même, si notre case est à la frontière de la grille alors elle n'a pas de lien sur cette frontière. Nous construisons donc la grille ligne par ligne, de haut en bas, de gauche à droite. Nous avons ci-dessous un exemple d'un niveau de 4 lignes et 7 colonnes.

0.0	0.1	0.2	0.3	0.4	0.5	0.6
1.0	1.1	1.2	1.3	1.4	1.5	1.6
2.0	2.1	2.2	2.3	2.4	2.5	2.6
3.0	3.1	3.2	3.3	3.4	3.5	3.6

Nous avons implémenté l'option où un nombre spécifique de composantes connexes peut être indiqué lors de la génération d'un niveau. Chaque fois qu'une pièce p est placée dans la grille, une composante connexe se forme, à laquelle appartient la pièce. Ensuite, nous parcourons les autres composantes connexes existantes, si l'une d'elle contient une voisine de p, cette dernière est supprimée après que la composante connexe de p ait ajouté

toutes ses pièces. Lorsque notre grille a fini d'être construite, nous comptons le nombre de composantes connexes restantes.

Pour pouvoir atteindre l'objectif, nous avons banni les pièces vides et favorisés les pièces de type 1 car deux pièces de ce type peuvent facilement former une composante connexe. Dès le début de la construction, nous essayons de construire le maximum de composantes connexes afin d'être certain d'atteindre le nombre de composantes connexes demandées. Ainsi, si on nous demande n composantes connexes. Nous formons n-1 composantes connexes avec des pièces de type 1. Ensuite nous utilisons toutes les autres pièces pour former la dernière composante connexe.

Le désavantage de cette technique est que le niveau est déséquilibré, toutes les composantes connexes sont en haut de la grille. Nous avons constaté que le nombre maximum de composantes connexes possibles pour un niveau est égale à la moitié du nombre de cases. Si le nombre demandé est supérieur, il est impossible de construire un niveau avec autant de composantes connexes, nous allons alors en faire le maximum : égale à la moitié des cases.

Un autre problème rencontré est que bien que le niveau soit construit avec le nombre de composantes connexes demandées, en faisant des rotations de pièces, le nombre change. Nous avons remarqué que le solver peut résoudre le niveau avec une solution différente. Donc le niveau généré par notre programme a plusieurs solutions, dont certaines ayant un nombre de composantes connexes différentes.

2. Vérificateur de solution

À l'aide du bouton 'Check' de l'interface graphique, ou en lançant la commande définie dans la classe main : 'java -jar phineloops-1.0-jar-with-dependencies.jar --check file', l'utilisateur peut vérifier si une grille est bien résolue ou non.

Afin d'assurer cette fonctionnalité, nous sommes partis de l'idée de balayer une grille, ligne par ligne, de gauche à droite. On vérifie si tous les liens que peut posséder une pièce, sont bien liés aux voisins (nord, est, sud, ouest) de cette dernière. Dès qu'un lien est perdu, l'algorithme retourne 'False'.

Pour simplifier la complexité de l'algorithme, tout d'abord, nous traitons les 4 cas où la pièce est à l'une des 4 bordures de la grille avant de vérifier les pièces du centre.

3. Solveur de niveaux

a) Choco solver

Nous avons décidé d'implémenter un premier solveur sous la forme d'un problème de satisfaction de contraintes (CSP) que nous avons étudié lors de l'UE Intelligence Artificielle. Pour cela nous nous sommes appuyés sur la librairie choco-solver-4.10.0.jar. Il nous fallait donc définir un modèle représentant les liens entre nos différentes pièces puis définir les différentes contraintes à respecter.

Notre première approche a été de définir pour chaque pièce 4 couples. Chaque couple représentant respectivement un lien ou non avec un voisin (nord, est, sud, ouest) et donc ayant une valeur 0 ou 1. Ainsi une pièce ayant une valeur 1 avec son couple nord par exemple signifiait qu'elle formait un lien avec son voisin nord. Cela permettait ainsi facilement d'implémenter les différentes contraintes. Pour une pièce, selon la valeur de son couple, cela obligeait son voisin à avoir la même valeur à son couple associé. Cette approche bien qu'assez intuitive à implémenter représentait un défaut majeur. En effet, chaque pièce possédait ainsi 4 variables à résoudre et demandait donc pour des grilles de grande taille, une allocation mémoire trop importante et ce solveur ne pouvait pas résoudre des grilles supérieures à 200x200.

Notre deuxième approche fut donc de modéliser chaque pièce par 4 bits. Pour chaque bit une valeur 0 signifiait que la pièce ne possédait pas de liens et inversement si sa valeur valait 1. Le premier bit représentant la direction nord, le deuxième la direction est, le troisième la direction sud et le quatrième la direction ouest. Ainsi pour un type de pièce donné et une orientation donnée chaque pièce possédait une valeur unique comprise entre 0 et 15.

Unicode	Type	Orientation	Valeur
	0	0	0
╵	1	0	1
╴	1	1	2
╷	1	2	4
╶	1	3	8
│	2	0	5
─	2	1	10
┴	3	0	11
├	3	1	7

┬	3	2	14
┤	3	3	13
┼	4	0	15
└	5	0	3
┌	5	1	6
┐	5	2	12
┘	5	3	9

Pour définir le domaine de chaque pièce de la grille, ses valeurs correspondent, selon son type aux différentes orientations possibles : une pièce étant “lockée” ou étant sur un bord ou un coin a un domaine d’orientations possibles réduit. Concernant les contraintes l’idée est similairement la même à celles de notre première approche. Pour une pièce donnée, selon la valeur de son bit à telle position, la pièce voisine associée doit donc avoir son bit correspondant à la même valeur. De même nous nous sommes ensuite rendu compte qu’en définissant seulement les contraintes sur les côtés est et sud d’une pièce cela suffisait à contraindre l’ensemble de nos pièces à respecter les conditions de liens ou non avec ses voisins. Grâce à cette approche, comme nous avons pu limiter le nombre de variables à résoudre ainsi que le domaine et les contraintes de celles-ci, notre solveur a pu ainsi résoudre de manière efficace de grandes grilles.

Pour intégrer le multithreading, nous nous sommes appuyés sur l’approche portfolio qui peut être intégré à un modèle défini par le choco solver. L’idée étant d’intégrer dans un portfolio plusieurs modèles identiques (correspondant au nombre de threads utilisés). Ainsi lors de la résolution du portfolio dès qu’un modèle est résolu, les autres s’interrompent et la solution est retournée. Néanmoins nous avons remarqué que cette approche du multithreading n’apportait que peu de gains de performance à notre solveur.

De même à partir de ce solveur, il a été facile d’ajouter une méthode nosolve permettant de résoudre une grille où chaque pièce n’est liée à aucun voisin. Pour cela nous avons juste modifié nos contraintes de liens entre deux pièces. En effet pour que deux pièces voisines ne soient pas liées il suffisait de spécifier que le produit de leurs bits associés prenne la valeur 0, la seule possibilité que ces deux pièces soient liées étant que leur bit respectif prenne la valeur 1.

Enfin via ce solveur, nous avons pu implémenter facilement une extension HEX où les pièces possédant 6 liens (nord, nord-est, sud-est, sud, sud-ouest, nord-ouest), les pièces pouvant faire deux liens avec leurs voisins situés à l’est ou à l’ouest de leur position. Ainsi pour le modèle du choco-solver chaque pièce est ainsi codée sur 6 bits à la place (annexe 1).

Pour cette extension nous avons seulement implémenté les classes PieceHex, GridHex et le solveur afin de montrer que la résolution s’effectuait mais nous n’avons pas

ajouté l'ensemble des méthodes et des autres classes permettant la visualisation d'un niveau, l'idée étant seulement de montrer que notre solveur pouvait s'adapter facilement pour résoudre des niveaux contenant ce type de pièces.

b) Stack solver

Nous avons décidé de créer un second solver suivant les conseils dans le sujet du projet. Tout d'abord, nous avons commencé par fixer les pièces qui ne pouvaient avoir qu'une seule orientation à cause de leur position dans la grille (bord ou case vide voisine). En les fixant, nous avons aussi pu réduire le nombre d'orientations de leurs voisins. Une pièce voisine n'ayant plus d'orientations possibles permettait de savoir que le niveau était faux. Cette technique a permis de résoudre les niveaux les plus simples. Toutefois il arrivait souvent qu'il reste dans la grille des pièces avec 2 à 4 orientations possibles. Les pièces encore mobiles ont alors été intégrées dans un arbre où les nodes étaient des pièces (inspiration : https://www.javagists.com/java-tree-data-structure).

Notre première tentative fut de faire un parcours en largeur, l'arbre ainsi construit avait chacune de ses branches représentait une solution réalisable du niveau. Toutefois cela prenait beaucoup de temps et nous n'avions besoin que d'une solution. Nous avons alors fait un parcours en profondeur et les résultats étaient bien meilleurs. Mais nous avons réalisés que ce n'était pas la meilleure structure car uniquement possible avec des petites grilles (<30x30) et que le sujet nous conseillait d'utiliser une pile.

Nous avons alors créé des piles contenant toutes les orientations possibles d'une pièce mobile. Chaque pile a été, elle aussi empilée dans une autre pile, cette dernière contenant toutes les pièces mobiles. Nous avons fait un parcours en profondeur de manière récursive des piles. Nous dépilions les piles pour trouver une solution réalisable, représentée par des nodes. Si ce n'était pas le cas, nous les empilions et réessayions avec une nouvelle pièce. Les résultats étaient remarquables car nous avons réussi à résoudre des grandes grilles (>100x100).

Nous avons aussi essayé de créer une unique pile avec toutes les pièces mobiles. Nous dépilions la pile et faisions des rotations des orientations possibles jusqu'à trouver une pièce réalisable sinon nous re-empilions. Les résultats furent décevants, nous n'avons jamais pu atteindre la résolution de grilles 100x100. Nous sommes donc revenus sur notre précédente méthode.

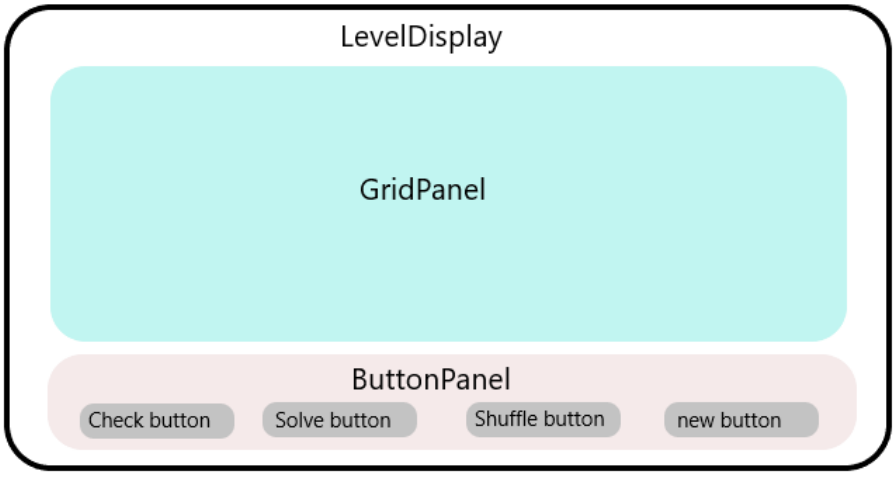
En ce qui concerne la pièce choisie, c'est celle qui a soit le minimum d'orientations possibles ou soit le maximum. Avec le minimum d'orientations possibles, nous avons eu de meilleurs résultats avec un thread. Le maximum est très intéressant avec l'utilisation de 4 threads mais devient aussi plus instable, risque de blocage (time out).

Nous avons aussi mis en place une méthode qui dépile en premier les pièces voisines à la pièce choisie, les résultats furent catastrophiques, nous avons donc gardés le dépilage de droite à gauche et de bas en haut.

Nous avons rencontré de grandes difficultés à implémenter les threads, nous avons eu des problèmes de synchronisation. Bien que nous l'employions dans notre programme, les threads ont rendu les résultats instables. La résolution d'un même niveau différait grandement d'une exécution à une autre à cause des threads : il résout en moins d'une seconde le niveau parfois et d'autres fois il reste bloqué sur le niveau. Pour ces raisons, nous préférons utiliser le choco-solver.

4. Visualisation de niveaux

Afin que l'utilisateur puisse jouer à infinity loop, nous lui avons créé une interface graphique. Cette dernière respecte le modèle MVC : modèle, vue et contrôleur. En effet, notre grille composée de pièces est notre modèle, la vue permet de visualiser notre modèle et le contrôleur permet à l'utilisateur d'interagir avec notre modèle via la souris et les boutons de la vue.

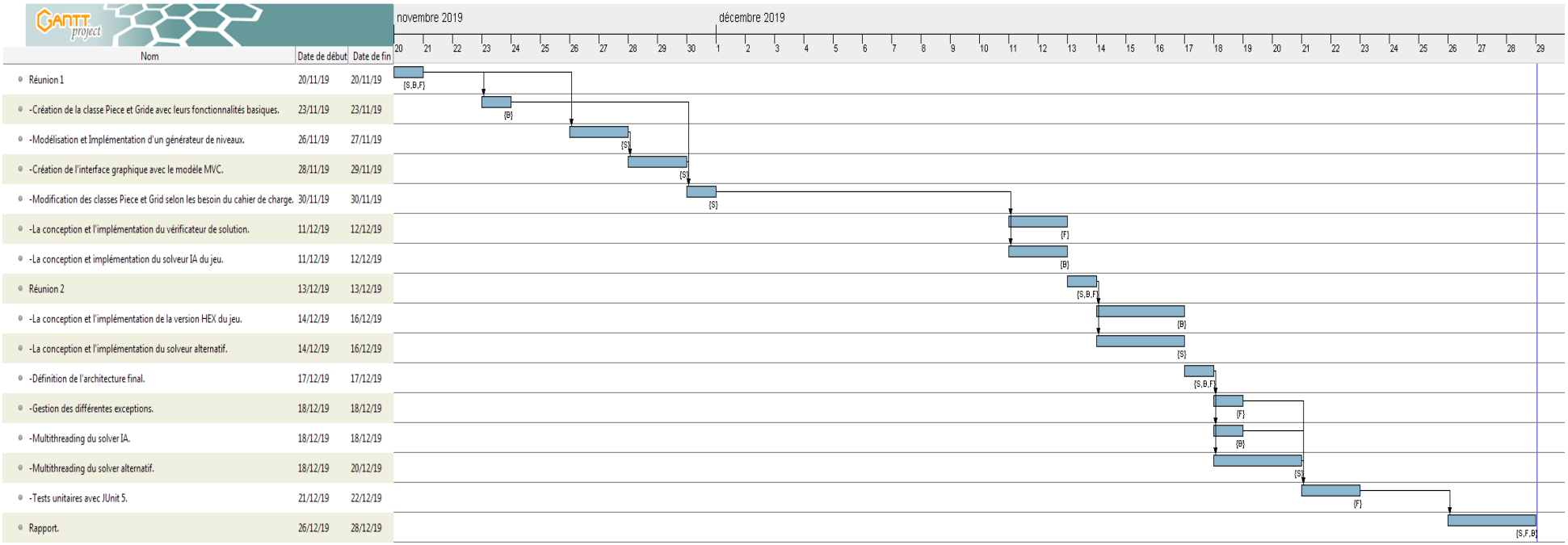


Nous avons eu quelques difficultés à faire l'interface graphique. Nous avions le choix entre deux librairies Swing et Javafx. Nous avions une préférence pour javafx due à la qualité et la clarté de ses dessins par rapport à Swing. Nous avons donc utilisé la librairie javafx mais il y a eu des problèmes lors de l'implémentation du contrôleur du pattern MVC : il était impossible de faire des rotations aux pièces. Nous avons alors décidé d'utiliser la librairie Swing avec laquelle nous étions plus familier. L'implémentation fut alors plus simple.

.

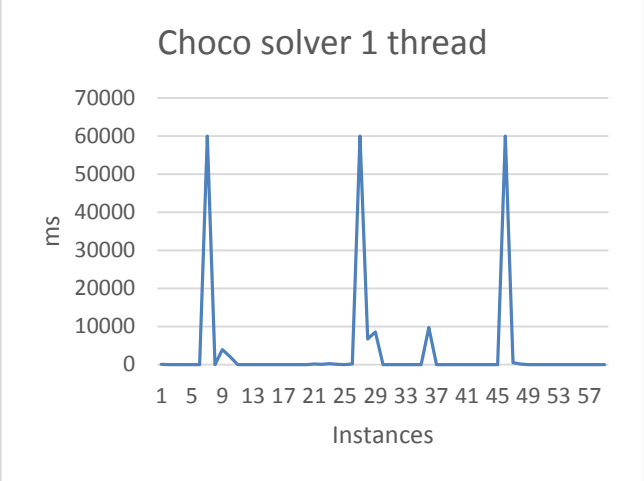
III. Déroulement et répartition des tâches

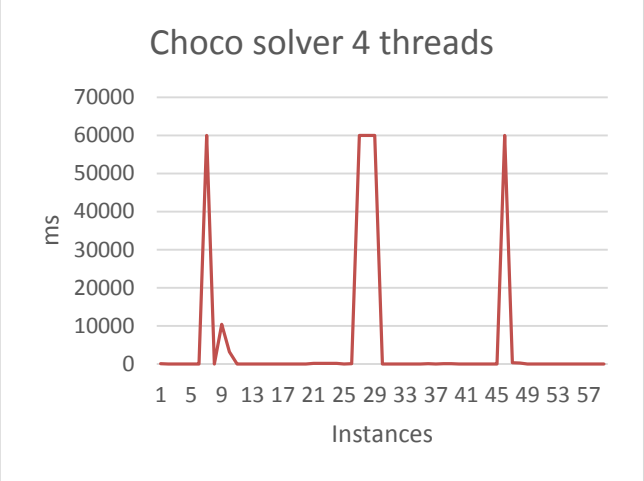
- La durée du projet : 20 nov. 2019 - 29 déc. 2019.
- Tâches : 16.
- Ressources : 3.

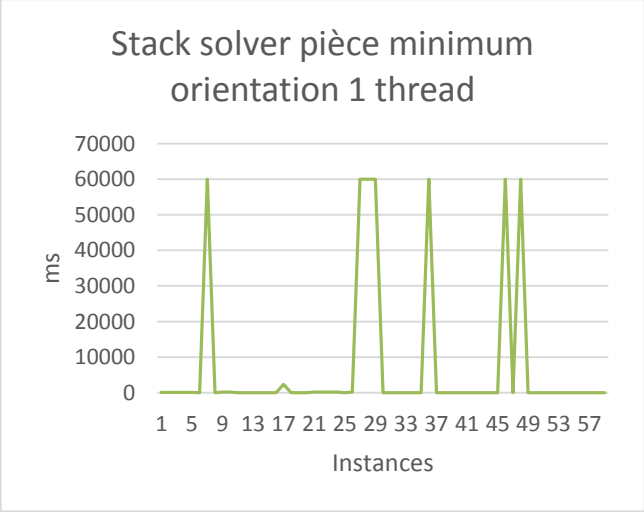


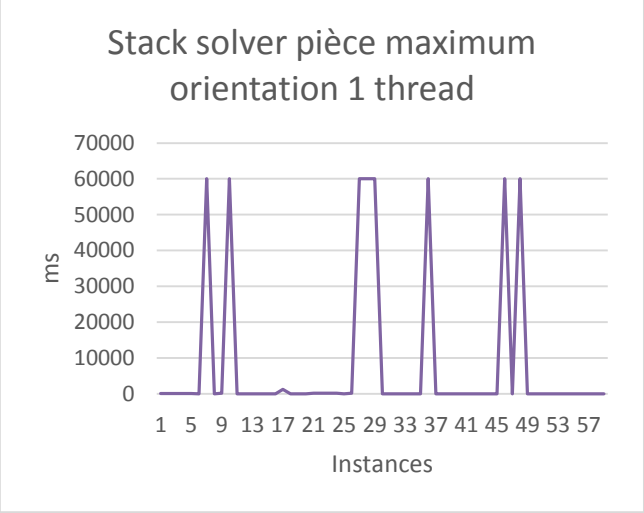
IV. Performances des solvers

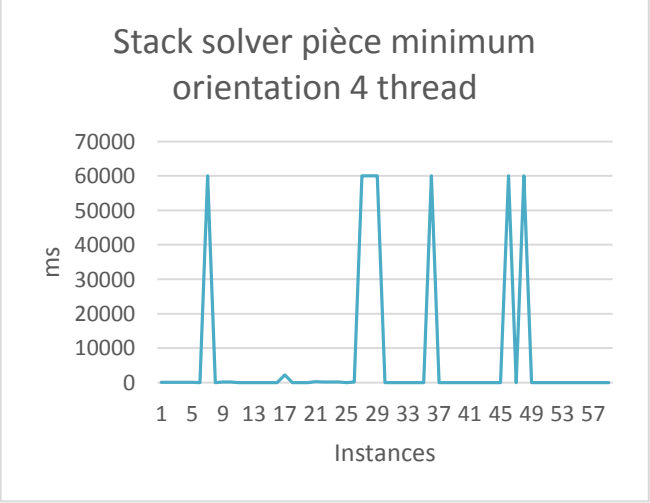
Nous avons testé nos solveurs sur les instances publiques. Une instance, ayant fait un time-out, est représenté par un pic à 60 000 millisecondes. Le solveur n'a pas réussi à la résoudre en moins d'une minute. Aucun de nos solveurs n'a réussi à résoudre toutes les instances publiques.

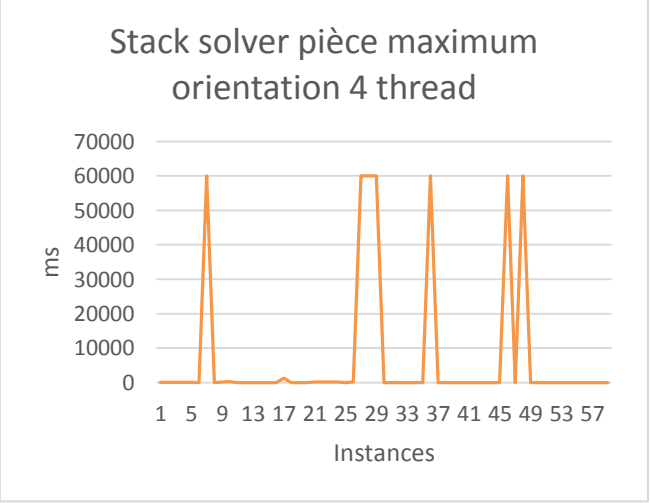


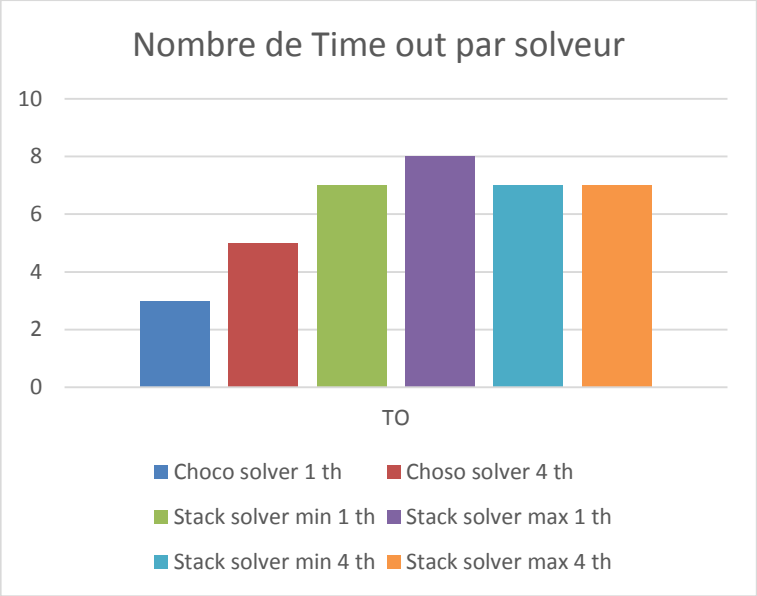








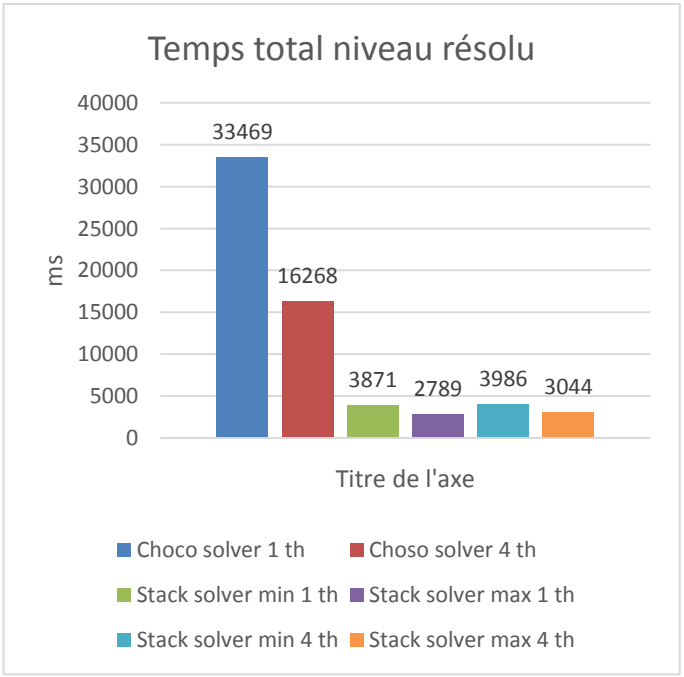


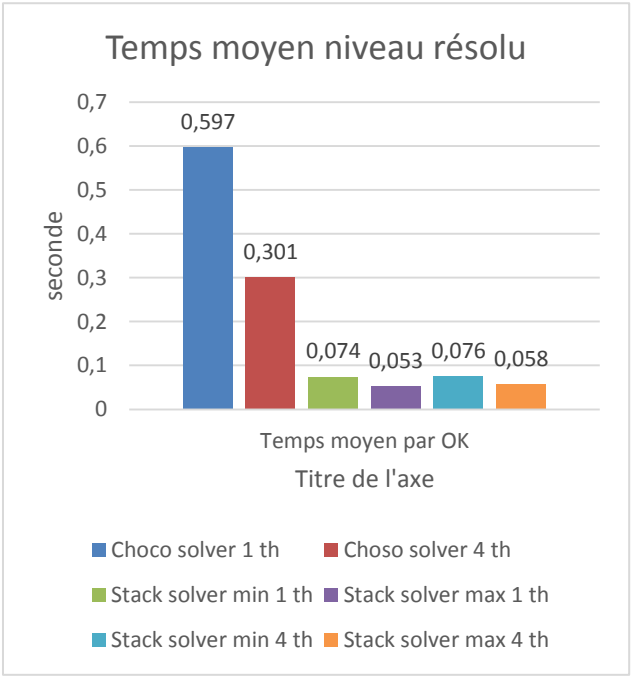


On remarque tout d'abord que le choco solver résout plus d'instances que le solver stack. De plus, le choco solver est plus performant avec un thread que quatre.

Quant au stack solver avec un thread, une pièce avec un minimum d'orientations est préférable à une avec un maximum car elle résout une instance de plus. Avec plusieurs threads, le résultat est similaire, toutefois le solver max est instable, car il lui arrive de ne pas résoudre la même instance(10ème) d'une exécution à une autre.

Le choco solver met plus de temps à résoudre car il résout des instances que le stack solver n'arrive pas, ces instances prennent plusieurs secondes à être résolus. De même, le solver stack max 1 thread met moins de temps à résoudre moins d'instances, donc il n'est pas intéressant de le comparer. Parmi les solveurs stack, le plus rapide est le solveur max avec 4 threads par rapport au solver min.





Conclusion

Nous avons réussi à générer des niveaux. Cela nous a permis ensuite de construire un vérificateur de solution. Par la suite nous avons programmé un solveur de niveau. Afin que l'utilisateur puisse jouer, nous avons créé une interface graphique.

Bien que nous ayons réalisé tout ce qui était attendu, notre application n'est pas optimale, les threads n'ont pas été gérés de la meilleure manière. De ce fait, nous n'avons pas réussi à résoudre toutes les instances sur la plateforme dans le temps imparti.

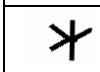
Pour conclure, nous sommes assez satisfaits du travail réalisé. Le bilan global est positif puisque notre application répond à la problématique de départ.

Annexes

Annexe 1 : Description des pièces de la version HEX

Unicode	Type	Orientation	Valeur
	0	0	0
	1	0	1
	1	1	2
	1	2	4
	1	3	8
	1	4	16
	1	5	32
	2	0	5
	2	1	10
	2	2	20
	2	3	40
	2	4	17
	2	5	34
	3	0	3
	3	1	6
	3	2	12
	3	3	24
	3	4	48

	3	5	33
	4	0	7
	4	1	14
	4	2	28
	4	3	56
	4	4	49
	4	5	35
	5	0	15
	5	1	30
	5	2	60
	5	3	57
	5	4	51
	5	5	39
	6	0	29
	6	1	58
	6	2	53
	6	3	43
	6	4	23

	6	5	46
	7	0	31
	7	1	62
	7	2	61
	7	3	59
	7	4	55
	7	5	47
	8	0	63
	9	0	9
	9	1	18
	9	2	36
	10	0	21
	10	1	42
	11	0	27
	11	1	54
	11	2	45
	12	0	11
	12	1	22

	12	2	44
	12	3	25
	12	4	50
	12	5	37
	13	0	41
	13	1	19
	13	2	38
	13	3	13
	13	4	26
	13	5	52