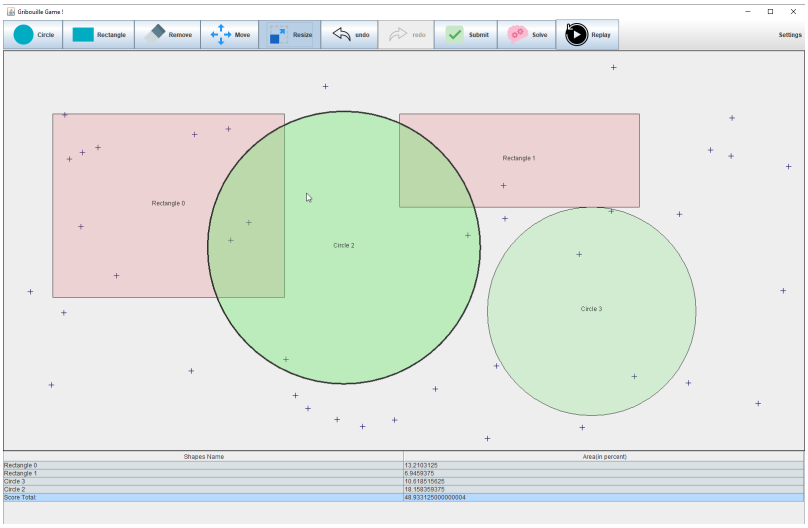
Université de Caen - Campus 2

Rapport de Projet

Patrons de conception avancés : Gribouille Game



Antoine GONZALEZ
(21504712)
Erwann LE ROUX (21500894)
Promo 2019 - 2020

Professeur : Yann MATHET

1er année de master Informatique.

Table des matières

Présentation du projet :

Dans le cadre du module *Patrons de conception avancés*, nous avons du réaliser une application Java dont le but est de nous faire travailler les divers patterns vus en cours. L'application est un jeu dans lequel l'utilisateur va devoir recouvrir des points (générés aléatoirement) dans un espace de dessin (canvas) à l'aide de formes qu'il peut peindre. Pour ce faire il dispose de divers outils communs à tout logiciel de dessin (outils de création, suppression, redimensionnement ...). Le joueur dispose d'un nombre limité de forme et doit optimiser la surface recouverte pour marquer des points. L'application est également dotée d'une intelligence artificielle pouvant proposer une résolution de l'instance de jeu. Un tableau récapitulatif permet de connaitre les surfaces occupées par les formes et le score actuel.

Dans ce rapport nous allons dans un premier temps aborder l'architecture de l'application, puis nous expliquerons un à un les différents patrons de conception implémentés, et finirons par aborder quelques algorithmes avant de conclure sur les perspectives d'amélioration.

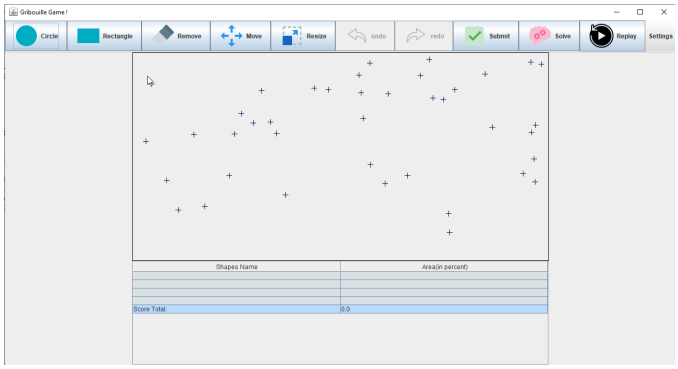


Figure 1 – Exemple de situation initiale.

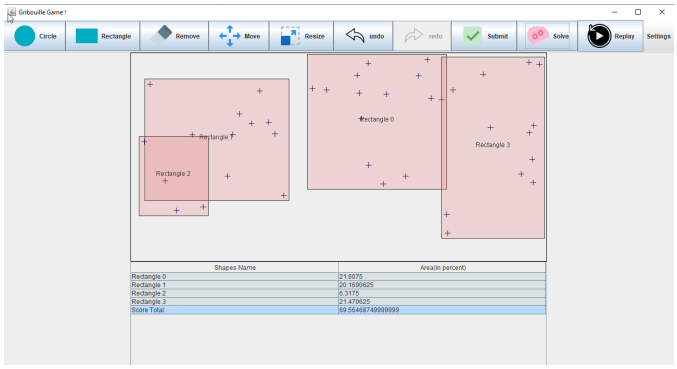


Figure 2 – Exemple de résolution.

Architecture de l'application :

L'application est basée sur une architecture MVC (modèle - vue - contrôleur). Ce type d'architecture divise nos fichiers sources en trois parties distinctes :

- Une partie **Model** : dans laquelle on retrouve les fichiers propres à **la modélisation des données**. On y trouve dans notre cas, les classes permettant de représenter notre instance de jeu et ses divers composants (entités géométriques, points, figures...)
- Une partie **View** : dans laquelle on retrouve les fichiers décrivant **la manière dont les données doivent être affichées à l'écran**. On y trouve dans notre cas, les classes représentant notre fenêtre de jeu et ses divers composant (GUI, JFrame, Canvas...).
- une partie **Controler** : dans laquelle on trouve les fichiers décrivant **comment les interactions utilisateurs doivent être interprétées** pour agir sur le modèle de données. On y trouve dans notre cas des "Listener", des classes décrivant le traitement à réaliser lorsque l'utilisateur interagit avec l'application (ex : MouseListener interprète les actions de la souris).

Notre application se divise donc en trois "packages" principaux pour chacunes de ces parties et un "package" supplémentaire *Utils* où on aura des classes utilitaires aux fonctionnalités variées

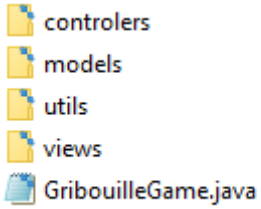


FIGURE 3 – Contenue de notre packgage src.

Le package models comporte :

- Une classes principale **GameModel.java** représentant l'état du jeu dans sa globalité (liste des points générés , listes des formes dessinées ...).
- Un sous-package **geometricEntities** contenant les classes permettant de décrire les divers entités géométriques et leur comportement (Circle.java , Point.java ...).
- Un sous-package **generateStrategy** et un sous-package **iaStrategy** contenant respectivement les classes décrivant les manières dont notre modèle peut générer les points initiaux et les manières dont l'IA peut résoudre le jeu.

Le package controlers comporte : les différents MouseListener décrivant la manière d'interpréter les actions souris de l'utilisateur.

Le package views comporte :

- Une classe principale **GUIGame.java** représentant l'interface utilisateur globale (la fenêtre de l'application).
- Une classe "vue/contrôleur" **Canvas.java** représentant l'espace du dessin au sein de la fenêtre. C'est par cette classe que les interactions utilisateurs sont captées et interprétées par délégation à ses contrôleurs associés .
- Une classe **statsModel.java** décrivant comment les données doivent être affichées dans le tableaux de scores/statistiques.
- Un sous-packgage **geometricEntitiesView.java** contenant les classes décrivant la manière de représenter chacune des entités géométriques de notre modèle (CircleView.java, PointView.java ...).

La mise en place de cette architecture nécessite l'utilisation de patrons de conception tels que le pattern MVC Observer/Observable. Patrons que nous allons maintenant expliciter.

Patrons de conception :

Mise en place MVC - pattern Observer/Observable :

La mise en place de pattern **MVC Observer/Observable** permet de faire interagir nos packages models, views et controlers tous en garantissant l'indépendance du modèle. Il introduit la notion d'objets observateurs et d'objets observables. **Un objet observateur va pouvoir s'abonner à un objet observable afin de pouvoir être notifié en cas de changement de ce dernier**.

1. Intéractions entre les classes GameModel et Canvas :

Dans notre application nous avons du mettre en place un pattern Observer/Observable afin de faire interagir notre canvas avec notre modèle de données. Pour ce faire nous avons défini deux interfaces : l'interface **GameModelObserver** implémenté par **Canvas.java** et l'interface **GameModelObservable** implémenté par **GameModel.java** [voir figure 4].

- **GameModelObservable**, définit des méthodes permettant à la classe observable de notifier ses observateurs d'un ajout ou d'une suppression d'une entité géométrique.
- **GameModelObserver**, définit des méthodes permettant à un observateur de se mettre à jour suite à la notification d'une classe observable (i.e lorsque celle-ci se voit ajouter ou retirer une entité géométrique).

Ainsi notre canvas seras averti par le modèle de données lorsque celui-ci se verra ajouter ou supprimer une entité géométrique, il pourra alors modifier son contenu afin de mettre à jour l'interface graphique.

Exemple de processus appelé lors d'une interaction de l'utilisateur :

1. L'utilisateur dessine un rectangle dans le canvas.
2. Le contrôleur capte les actions souris.
3. Le contrôleur traduit les actions souris et ajoute un rectangle au modèle de jeu.
4. Le modèle de jeu modifié notifie le canvas qu'on lui a ajouté un rectangle.
5. Le canvas notifié créer et dessine la vue du rectangle.

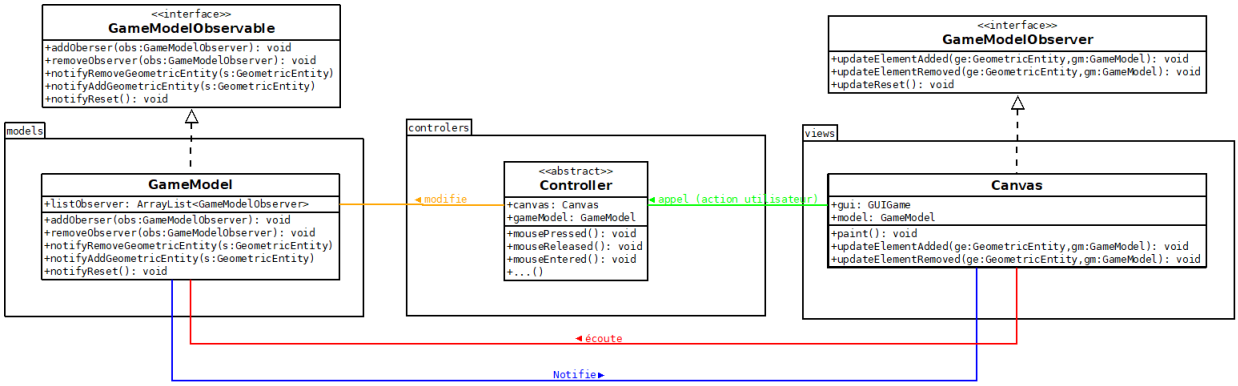


FIGURE 4 – Diagramme de classe représentant les interactions entre GameModel et Canvas.

2. Intéractions entre les classes GeometricEntity et View :

De même nous faisons interagir chaque entité géométrique du modèle de donnée avec sa vue singulière. Ainsi lorsqu'une forme est déplacée (ou modifiée) la vue associée est alors notifiée et modifiée, elle notifie alors le canvas pour être repeinte. Pour cela nous définissons deux autres interfaces [voir figure 5] :

- **Observable** qui définit des méthodes permettant à la classe observable de notifier ses observateurs
- **Observer** définit des méthodes permettant à un observateur de se mettre à jour suite à la notification d'une modification.

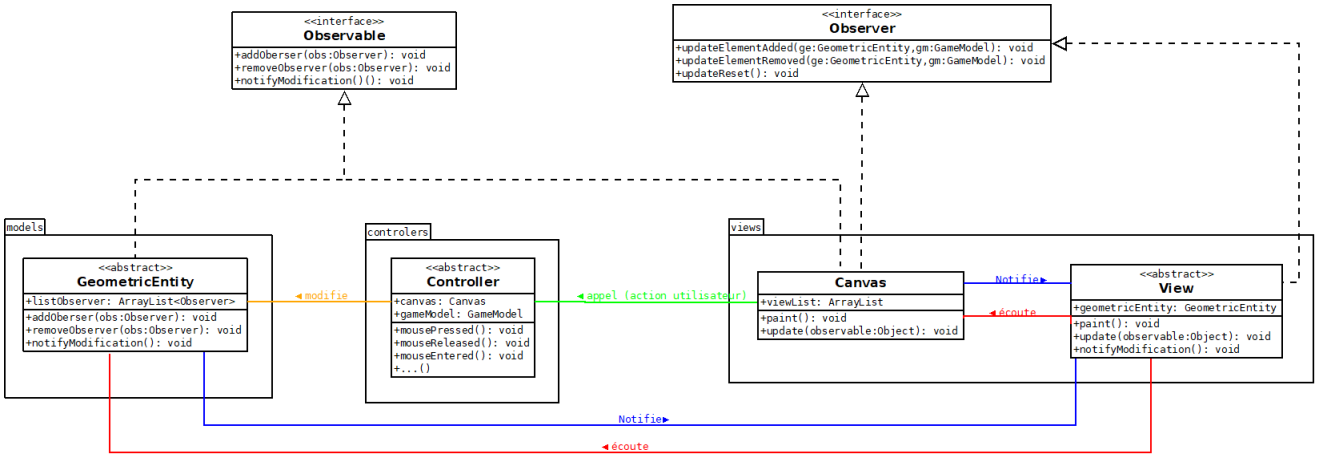


FIGURE 5 – Diagramme de classe représentant les interactions entre GeometricEntity et View.

Divers contrôleurs - pattern State :

L'utilisateur a à disposition plusieurs fonctionnalités lui permettant d'interagir avec le canvas, il peut par exemple dessiner, supprimer ou redimensionner une forme. Le canvas va donc **devoir interpréter différemment les actions souris de l'utilisateur en fonction du mode de dessin** dans lequel il est. Pour cela nous devons définir **différents types de controler** pour interpréter les actions souris (MouseListener). Nous avons mis en place un **pattern State** pour changer le type de controleur de notre canvas en fonction du mode de dessin choisi par l'utilisateur. **L'objectif est donc de donner à notre canvas un comportement variable**.

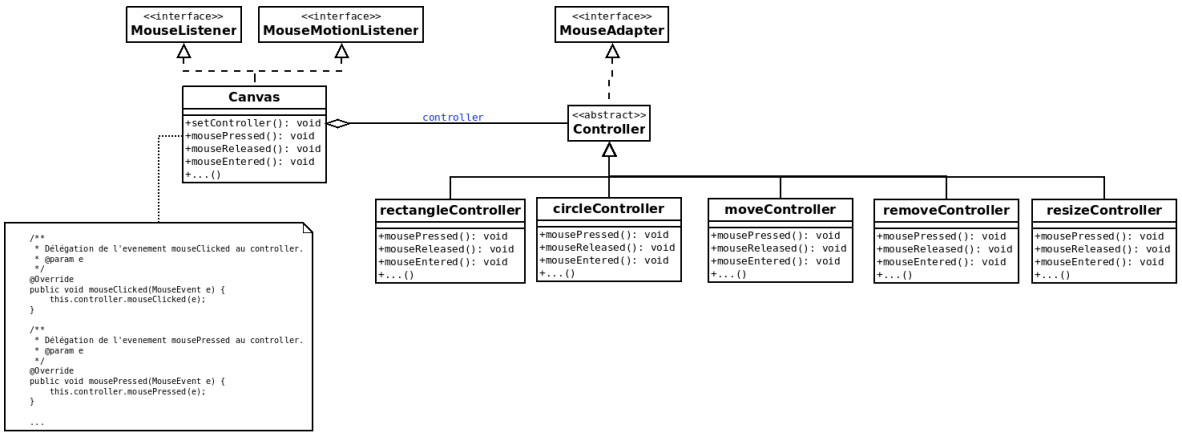


FIGURE 6 – Diagramme de classe représentant le pattern State mis en place.

Ainsi lorsque l'utilisateur changeras de mode de dessin via le menu suivant :

Nous positionnerons son controler dans le bon "Etat" à l'aide de la méthode **setController()** de notre classe Canvas.java.
Exemple : Si l'utilisateur choisit l'outil permettant de déplacer une forme nous appelons :

canvas.setController(new moveController());

Boutons undo/redo - pattern Command :

Notre Application permet à l'utilisateur de **pouvoir annuler et rétablir une action** à l'aide d'un système de boutons **undo/redo** :

Pour mettre en place cette fonctionnalité nous avons du implémenter **le pattern Command** [voir figure 7]. Celui-ci **permet de découpler l'instanciation d'une action utilisateur de son exécution**. Ainsi il est possible de créer des actions et de pouvoir les exécuter et les annuler plus tard lors du fonctionnement de l'application.

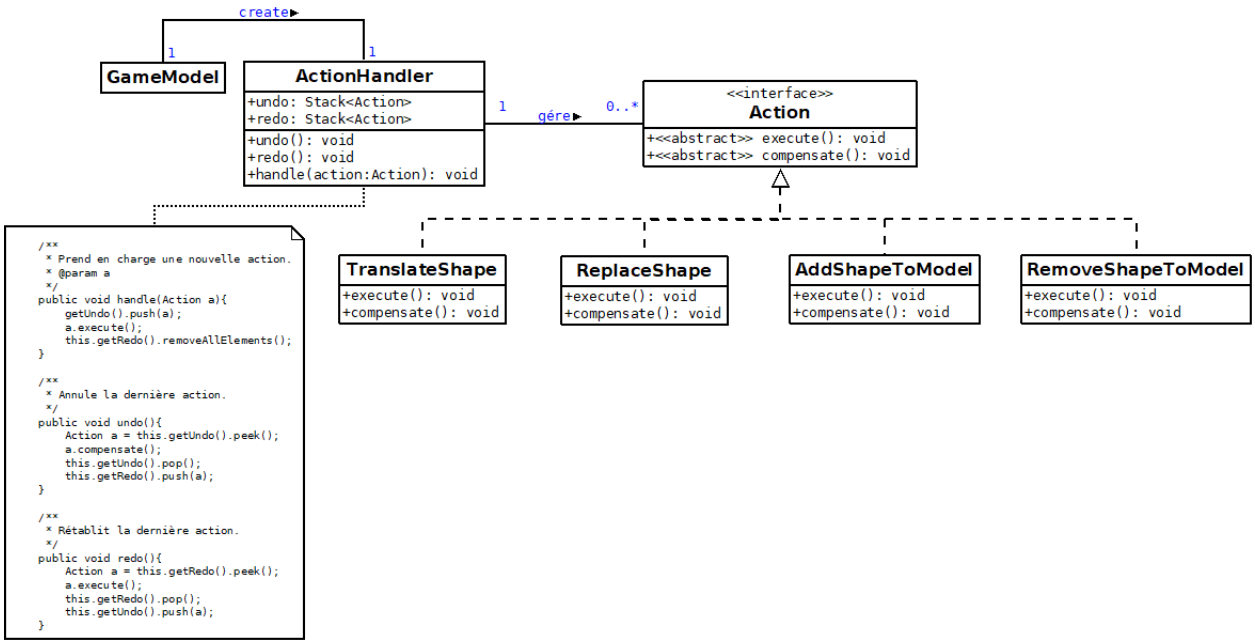


FIGURE 7 – Diagramme de classe représentant notre implémentation du pattern Command.

On y trouve :

- La classe abstraite **Action.java** qui permet de représenter une action utilisateur, elle définit deux méthodes : **execute()** et **compensate** respectivement appelées pour exécuter et annuler l'action.
- Des implémentations concrétes d'action qui dérivent de **Action.java** comme par exemple la classe **Translate.java** qui représente une action de translation de forme.
- La classe **ActionHandler.java** qui va gérer les actions de notre applications en les stockant dans les piles **undo** ou **redo**. Elle déclare trois méthodes :
  1. **handle(Action action)** : qui permet de prendre en charge une action (l'exécute).
  2. **undo()** : qui permet d'annuler la dernière action exécutée.
  3. **redo()** : qui permet de rétablir la dernière action annulée.

Actions - pattern Composite :

Conjointement au pattern commande nous avons utilisé le **pattern Composite** ain de pouvoir utiliser un groupe/une séquence d'actions comme si il s'agissait d'une action unique. C'est à dire **pouvoir annuler et rétablir une séquence entière d'actions**. Pour cela nous avons implémenté la classe **ActionComposite.java** [voir figure 8].

Cette classe entretient une liste d'instances d'objets Action tout en en étant elle même une. (Nous n'utilisons pas cette classe dans le projet mais l'avons implémenter en prévision d'une éventuelle utilisation.

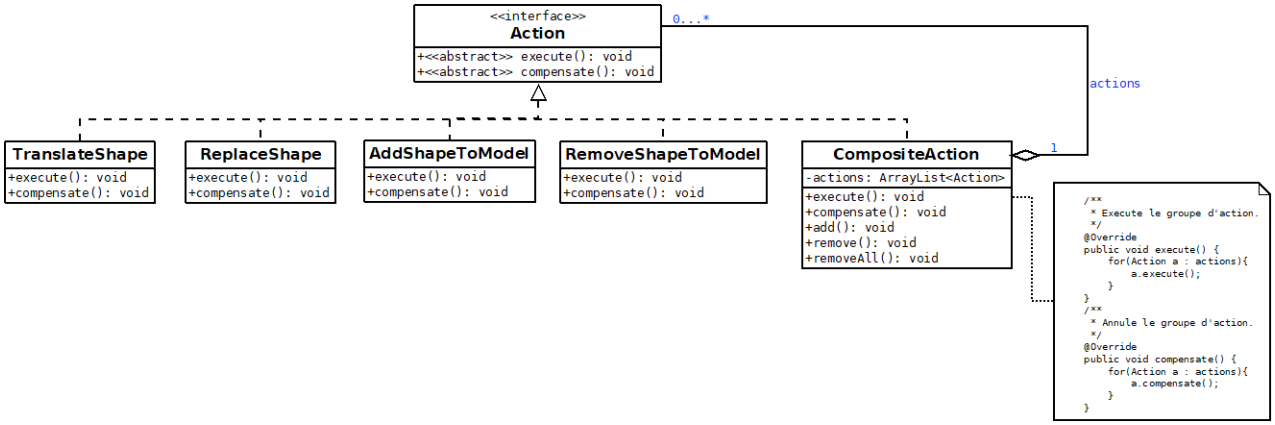


FIGURE 8 – Diagramme du pattern Composite implémenté.

View Loader - pattern singleton :

La classe ViewLoader que nous avons créé va nous servir à maintenir un niveau d'abstraction élevé lors de l'association de nos formes à leurs vues respectives et par conséquent éviter les enchainements de « if forme instance of ». Un des gros avantage de cette technique est que ça nous permettra par la suite d'ajouter une nouvelle forme très facilement.

Nous avons donc mis en place cette classe utilitaire en respectant le pattern Singleton pour s'assurer d'avoir une unique instance de celle-ci.

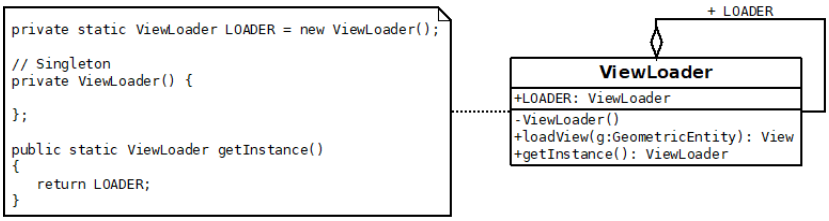


FIGURE 9 – Diagramme du pattern singleton.

Pour utiliser notre ViewLoader il suffira donc simplement d'utiliser ViewLoader.getInstance() puis d'utiliser la méthode loadView sur l'instance récupérée avec en parametre une instance de GeometricEntity qui nous renverra la vue associée.

Par exemple :

GeometricEntity g = new Circle()

View v = ViewLoader.getInstance().loadView(g)

Ce chargement dynamique des classes est possible grâce aux conventions de nommage que l'on utilise. Typiquement dans notre exemple, on récuperera la classe effective de notre entité soit « Circle » pour charger la classe « CircleView » et retourner directement une instance de « CircleView ». Nous avons réalisé cette classe en prévision d'intégrer les plugins à notre application car dans l'hypothèse ou on voudrait ajouter une forme dynamiquement, tous les « instance of » auraient posé problème tandis que notre ViewLoader aurait parfaitement assuré cette tâche.

Génération de point - pattern Stratégie :

Nous avons utilisé le pattern Stratégie pour pouvoir faire varier la manière dont les points vont être générés en début de partie. Notre modèle définira alors plusieurs algorithmes de génération de point que l'on pourra faire varier pour obtenir le comportement souhaité. Ces algorithmes sont encapsulées dans différentes implémentation de la classe abstraite **PointGenerator.java**.

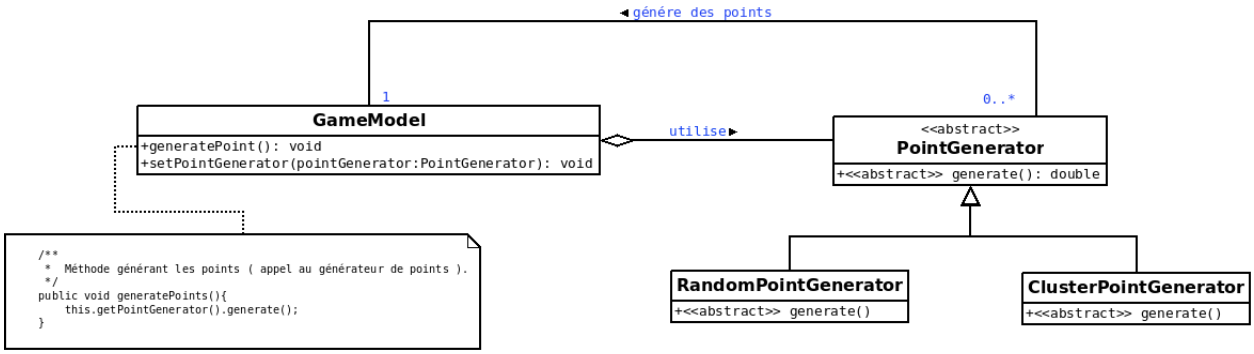
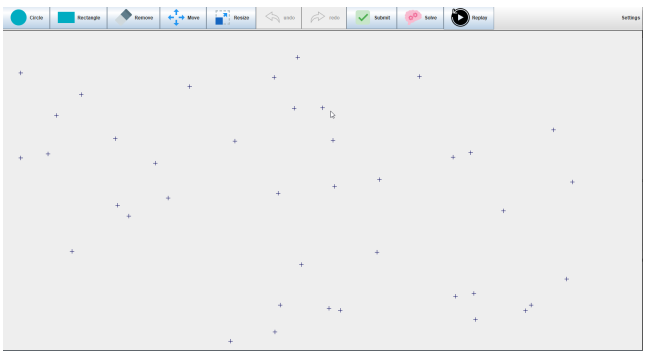


FIGURE 10 – Diagramme de classe représentant le pattern Stratégie utilisé pour la génération des points initiaux.

La classe **GameModel.java**, qui sert de **contexte** définit une méthode **generatePoint()** qui **délègue la génération** à la stratégie/l'algorithme sélectionné. La stratégie à adopter peut être modifiée au cours de l'exécution via l'appel à la fonction **setPointGenerator()** de cette même classe.

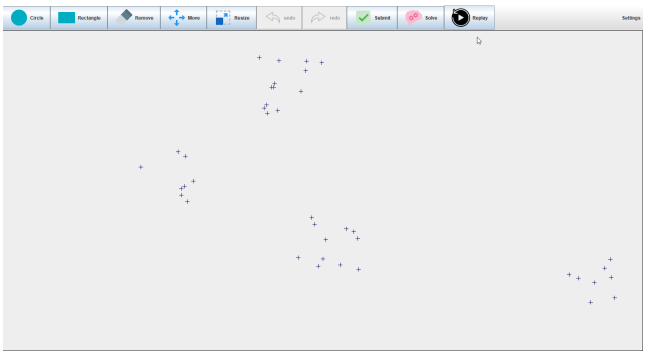
La première implémentation, **RandomPointGenerator.java**, définit un algorithme de génération aléatoire des points au sein du canvas.

Exemple :



La seconde implémentation, **ClustersPointGenerator.java**, définit un algorithme de génération basé sur l'utilisation de cluster. L'algorithme tire k points aléatoires (centre des clusters) autours desquels il tire les points initiaux.

Exemple :



Résolution - pattern Stratégie :

Nous utilisons également le pattern Stratégie pour pouvoir faire varier la manière dont l'IA va résoudre l'instance de jeu. Notre modèle définira alors plusieurs algorithmes de résolution que l'on pourra faire varier pour obtenir le comportement souhaité. Ces algorithmes sont encapsulés dans différentes implémentations de la classe abstraite **GameResolver.java**.

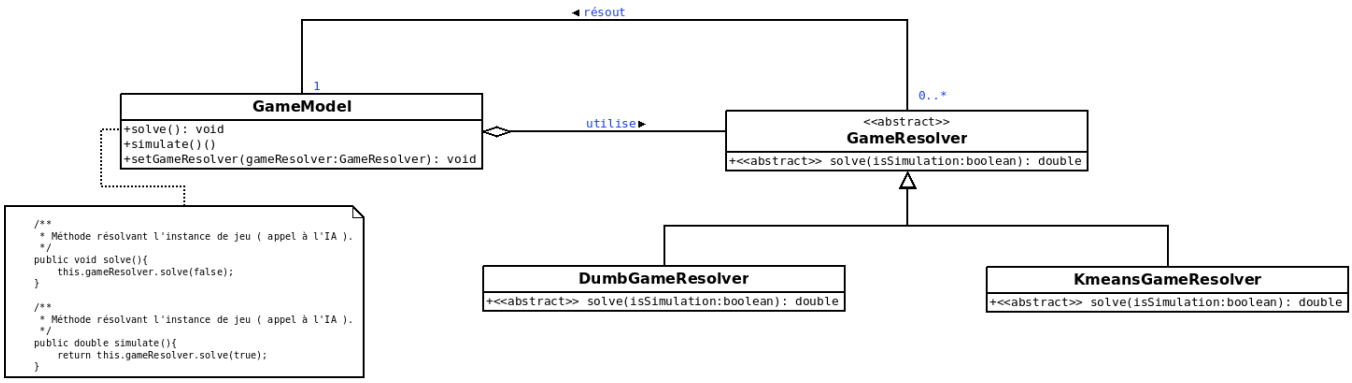
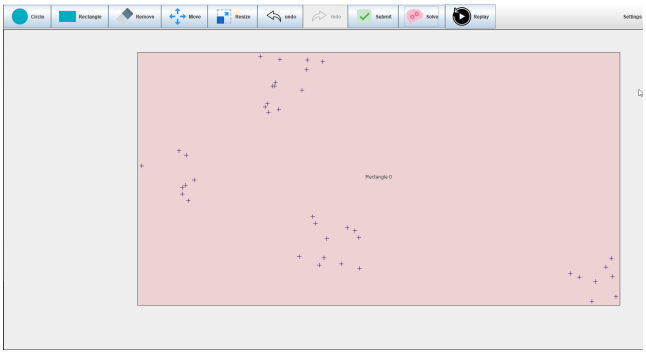


FIGURE 11 – Diagramme de classe représentant le pattern Stratégie utilisé pour la résolution du jeu par l'IA.

La classe **GameModel.java**, qui sert de **contexte** définit une méthode **solve()** qui **délègue le traitement de la résolution** à la stratégie/l'algorithme sélectionné. La stratégie à adopter peut être modifiée au cours de l'exécution via l'appel à la fonction **setGameResolver()** de cette même classe.

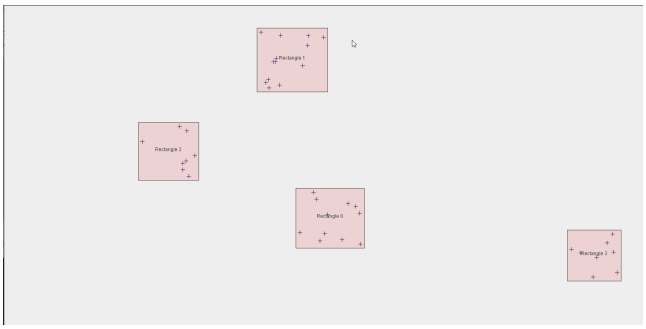
La première implémentation, **DumbGameResolver.java**, définit un algorithme de résolution de bas niveau où l'IA décide de dessiner un énorme rectangle recouvrant l'ensemble des points.

Exemple :



La seconde implémentation, **KmeansGameResolver.java**, définit un algorithme de résolution de haut niveau basé sur la génération de cluster que nous détaillerons plus tard.

Exemple :



Algorithmes et particularités du code :

Dans cette partie nous allons voir certaines particularités de notre version de l'application avec notamment l'explication de certains algorithmes.

Algorithme des k-moyennes :

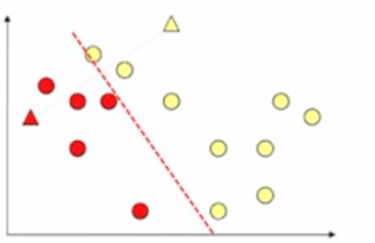
Pour avoir une intelligence artificielle de plus haut niveau nous avons utilisé **l'algorithme des k-moyennes**, il permet de **diviser une population de points en k groupes (clusters) de manière à minimiser la distance des points au sein d'un groupe**. Une fois les clusters générés notre IA n'a alors plus qu'à recouvrir chacun d'eux par une forme pour avoir une résolution convenable du jeu.
Voici notre implémentation de k-moyenne détaillée étape par étape :

```
public ArrayList<ArrayList<Point>> kMeans(int k, ArrayList<Point> population, int xMin, int xMax,int yMin, int yMax){
    boolean convergence = false;
    HashMap<Point,Integer> map =  new HashMap();
    ArrayList<Point> centroids = new ArrayList();
    ArrayList<ArrayList<Point>> clusters;
    // on tire aléatoirement k centroids (centre de cluster).
    centroids = pullCentroids(k, xMin, xMax, yMin, yMax);
    // on associe chaque point de la population à son centroid le plus proche.
    clusters = computeClusters(population, centroids, map);
    // tant qu'il n'y a pas convergence ( i.e les clusters ne sont pas fixes).
    while(!convergence){
        // on recalcule la position des centroids.
        centroidsRecompute(clusters,centroids);
        HashMap<Point,Integer> oldMap= (HashMap<Point,Integer>) map.clone();
        // on re-associe chaque point de la population à son centroid le plus proche.
        clusters = computeClusters(population, centroids, map);
        // on regarde si les populations des clusters ont changées. Si oui alors il n'y a pas convergence sinon il y a converfence.
        for(Entry<Point, Integer> entry : map.entrySet()) {
            convergence = true;
            Point Point = entry.getKey();
            Integer centroidsId = entry.getValue();
            if(oldMap.get(Point) != centroidsId){
                convergence =false;
                break;
            }
        }
    }
    return clusters;
}
```

FIGURE 12 – Fonction kmeans implémenté.

Initialisation :

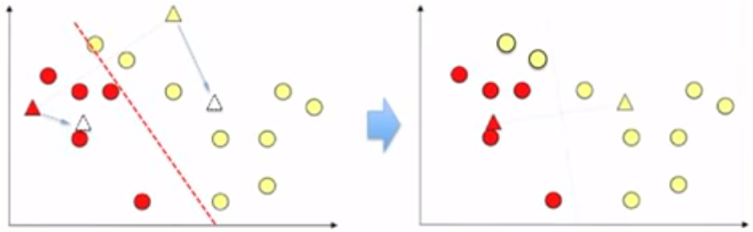
- 1 - nous tirons k centroids aléatoirement avec la méthode **pullCentroids(...)** , un centroid est un point qui représente le centre d'un cluster.
- 2 - nous associons chaque point à son centroid le plus proche avec la méthode **computeCluster(...)**. Cela revient à faire appartenir un point à un cluster.



Exemple d'initialisation pour k = 2 (triangle = centroid)

Corps de boucle :
Nous répétons les étapes suivantes :

- 1 - On repositionne chaque centroid au centre de son groupe en calculant la position moyenne des points constituants le groupe, avec la méthode **centroidsRecompute()**
- 2 - On ré-associe chaque point à son centroid le plus proche avec la méthode **computeCluster(...)**



Exemple d'itérations (repositionnement et ré-association).

La boucle s'arrête lorsqu'il y a convergence c'est à dire que la composition des clusters reste la même d'une itération à une autre.

Une fois que l'IA a généré les clusters avec l'algorithme des k-moyennes elle dessine un rectangle pour chaque, on obtient donc ce genre de résolution :

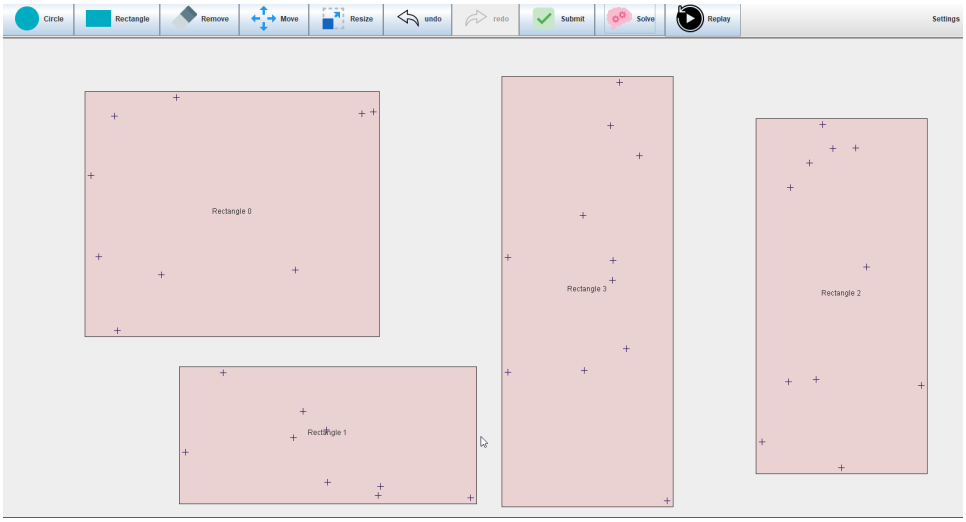


FIGURE 13 – Exemple de résolution avec la stratégie Kmeans.

ReplaceAction :

Lorsque que l'utilisateur modifie une forme nous procédons d'une manière particulière dans la manière de modifier notre modèle de données. En effet nous ne modifions par directement l'instance de forme (classe Shape.java) concerné mais une copie de cette forme. Une fois la copie modifiée nous permutons l'instance de la forme par sa copie. Ainsi pour annuler l'action il suffit de permuter de nouveau les deux instances. Le canvas repeint la forme modifiée une fois qu'il est notifié de l'ajout de la copie au modèle grâce à la méthode updateElementAdded() [voir MVC figure 4].

On a donc mis en place une classe ReplaceAction.java, implémentant Action.java.

Cette manière de faire pose une limite dans notre conception, en effet lorsqu'une forme est modifié nous devons alors recréer une vue pour la copie modifiée. Une amélioration serait d'utiliser le pattern Observer / Observable évoqué plus haut [voir figure 5] pour directement modifié la forme concerné et repeindre sa vue associée sans passé par une copie.

Conclusion :

Pour conclure, ce projet nous a permis d'approfondir certains patterns abordés l'année dernière en Licence (ex : Stratégy , MVC ...) tout en en apprenant de nouveaux tels que les patterns Command et State. Dans le but d'améliorer l'application nous aurions aimé mettre en place l'implémentation des plugins permettant par exemple d'ajouter de nouveaux types de formes ou de stratégies pour l'IA. Malheureusement, la charge de travail globale de ces dernières semaines (toutes matières confondues) ne nous a pas permis d'aboutir à cet objectif. Nous avons cependant amélioré certains "problèmes" évoqués lors de votre passage dans les groupes, à savoir :

- Améliorer l'interface pour visualiser les formes survolées par la souris (dans le cas où plusieurs formes se superposent).
- Éviter certaines redondances de code (voir méthode solve() et simulate() des GameResolver).
- Différencier le score dans la JTable à l'aide de style (couleur).
- Instancier le contrôleur (Controler.java) au niveau du canvas (Canvas.java) et non au niveau de la JFrame (GUIGame.java).