
Université de Caen Normandie

Patrons de conception avancés

M1 - IA, Science des Données et Santé & M1 Informatique à la carte

Année 2023 - 2024

---

# Développement d'un logiciel de dessin utilisant les patrons de conception

---

Projet réalisé par :

ANDRES Romain
MESSILI Islem
LADUREE Luca
OROU-GUIDOU Amirath Fara

Sous la supervison de :

MATHET Yann

Second semestre 2023-2024

# Table des matières

# 1 Introduction

Le but de ce dernier est de réaliser, en groupe, un logiciel de dessin de formes avec interface graphique, en intégrant un maximum de pattern vus en cours et en respectant le concept réalisé du modèle **MVC**(**M**odèle **V**ue **C**ontrôleur). Les grandes lignes de ce projet étant :

- de développer un modèle complètement indépendant de la partie graphique ;
- une partie vue-contrôleur venant se greffer sur le modèle ;
- de développer une application qui comporte un panel dans lequel l'utilisateur peut créer des formes ;
- de développer une interface qui disposera d'une barre de boutons permettant de choisir le mode actuel :
  - création d'un cercle
  - création d'un rectangle
  - déplacement d'une forme
  - redimensionnement d'une forme
  - suppression d'une forme

  Elle pourra aussi disposer d'un tableau (JTable) indiquant sur chaque ligne le nom d'une forme utilisée ainsi que sa surface
- d'ajouter des boutons « undo-redo ».

# 2 Logiciel de dessin de formes

## 2.1 Généralités

L'implémentation d'un tel logiciel de dessin implique naturellement la manipulation de formes géométriques. On comprend donc la création, la suppression, le déplacement, la coloration, ainsi que le redimensionnement de carrés, cercles et lignes. Notre application dispose de deux modes : un mode de dessin libre et un mode de jeu.

## 2.2 Dessin libre

L'objectif du mode de dessin libre est assez intuitif : il présente toutes les fonctionnalités présentées précédemment et permet donc à l'utilisateur de manipuler autant de formes qu'il le souhaite. Il peut également annuler une action avec les boutons undo/redo, et sauvegarder une image.

## 2.3 Jeu

Dans le mode jeu, des formes obstacles (en rouge) sont générées et placées aléatoirement dans la fenêtre de jeu. Le but est alors de placer un nombre limité de rectangles et de cercles afin de maximiser la surface qu'elles occupent en moins de 10 secondes. Ensuite l'ordinateur va donner sa solution. Le joueur gagne si il a fait une meilleur performance. On peut choisir la difficulté du solveur au début du jeu entre facile et médium. Pour l'instant le solveur par défaut est un solveur qui génère des formes aléatoires beaucoup de fois et garde la meilleur version. Ainsi plus le nombre d'itérations est grand, meilleur sera la solution. Une métrique a aussi été définie qui permet de définir le facteur d'agrandissement des formes pendant la phase itérative. Mettre 2 va vite créer des collisions et donc avoir des petites formes alors que mettre 1.1 va croître petit à petit et donc favoriser une meilleur solution au détriment des coûts en calculs pour élevés.

# 3 Organisation du projet

## 3.1 Répartition des tâches

Le projet étant essentiellement composé de plusieurs parties, nous nous sommes dans un premier temps concerté sur une modélisation UML, ce qui nous a permis de comprendre différentes façons d'implémenter notre logiciel. Ainsi Romain s'est chargé de créer le modèle de géométrie, les tests unitaires, à créer un solveur avec différentes stratégies et gérer les collisions dynamiques, Islem s'est chargé de tout l'aspect interface graphique avec la gestion des menus, des objets swing, a fait le pattern State et une partie du controler,

Amirath s'est chargé de faire le pattern command pour le undo et redo avec Luca, et Luca quant à lui créer un Adapter pour convertir un objet Container en JTable, codé une partie du Controller, ainsi que le pattern Factory.

### 3.2 Architecture du projet

Concernant l'architecture de notre projet, nous avons voulu la relier au sujet pour une meilleure compréhension. Ainsi nous avons opté pour une décomposition en trois packages :

- **Modèle** : permet d'implémenter le jeu de façon indépendante.
- Contrôleur : permet de gérer tout ce qui est évènement ;
- Vue : permet de gérer tout ce qui est rendu graphique. Elle se complète avec le contrôleur.

## 4 Les patterns utilisés

Pour ce projet, nous avons implémenter les patterns les plus adaptés aux différents cas de figure que nous avons pu rencontrer au cours du développement :

### 4.1 Pattern MVC

Comme vu précédemment, le modèle MVC est considérer comme un pattern structurel. Il permet d'organiser le code source autour de trois piliers :

- Le **Model** qui représente la structure des données. Leur définition ainsi que les fonctions qui leur sont propres et qu'elles peuvent avoir. Ce module est complètement décorrelé du code métier ou de l'affichage de telle sorte à ce que la modification de la logique ou de l'interface n'affecte pas la structure des données.
- La **View (ou les vues)** représente l'interface graphique à livrer au client qui en fait la requête. Avoir le code lié à l'interface isolé de la logique métier ou des données permet de faire des modifications à l'interface graphique sans avoir à se soucier de casser du code métier ou la structure des données.
- Le **Controller** sont au cœur de la logique métier de l'application. Ils se situent entre les vues et le model. Les requêtes qu'un client va faire depuis l'interface graphique, la view, vont être dirigées vers un controller qui sera en charge de manipuler les données dont il a besoin avec la brique Model, la traiter suivant le besoin métier, puis ordonner à la view de répondre au client avec les bons éléments.

Diagramme UML

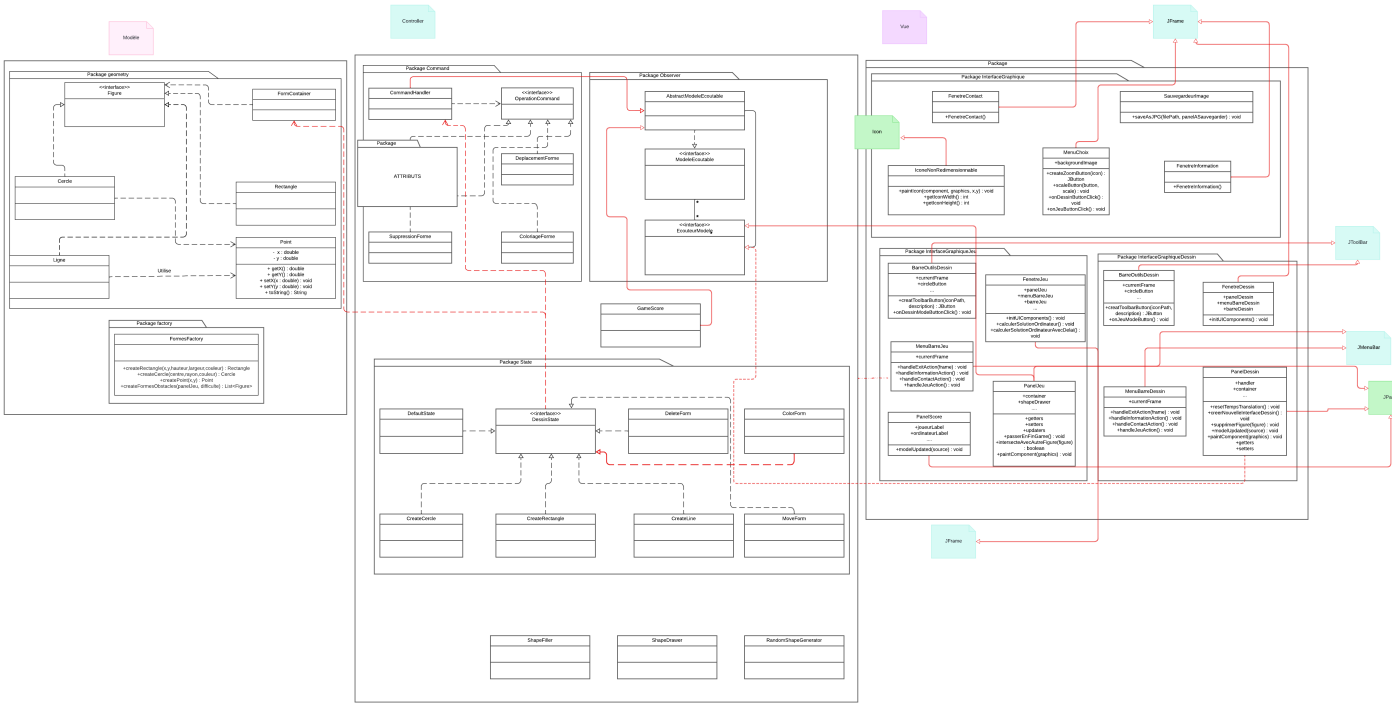

FIGURE 1 – Diagramme du pattern MVC

Ainsi nous utilisons le pattern MVC un peu partout dans le projet et c'est le pattern le plus essentiel, il met à jour toutes les vues au bon moment.

## 4.2 Pattern Observer

L'application se base sur le modèle MVC qui a deux principes essentiels à respecter :

- Le modèle doit être complètement indépendant de la vue contrôleur.
- La vue doit être à jour à chaque modification dans le modèle (le modèle prévient la vue), d'où l'utilisation du **pattern Observer**.
- Nous avons utilisé deux interfaces qui représentent deux types d'évènements à écouter :
  - **EcouteurModele** : cette interface permet la vue d'écouter tout changement dans le modèle.
  - **ModeleEcoutable** : cette interface permet la vue d'écouter les changements du conteneur des formes.

Le pattern Observer nous a permis d'avoir un modèle totalement indépendant de la vue et en même temps mettre à jour cette dernière en fonction des manipulations du joueur sur les formes.

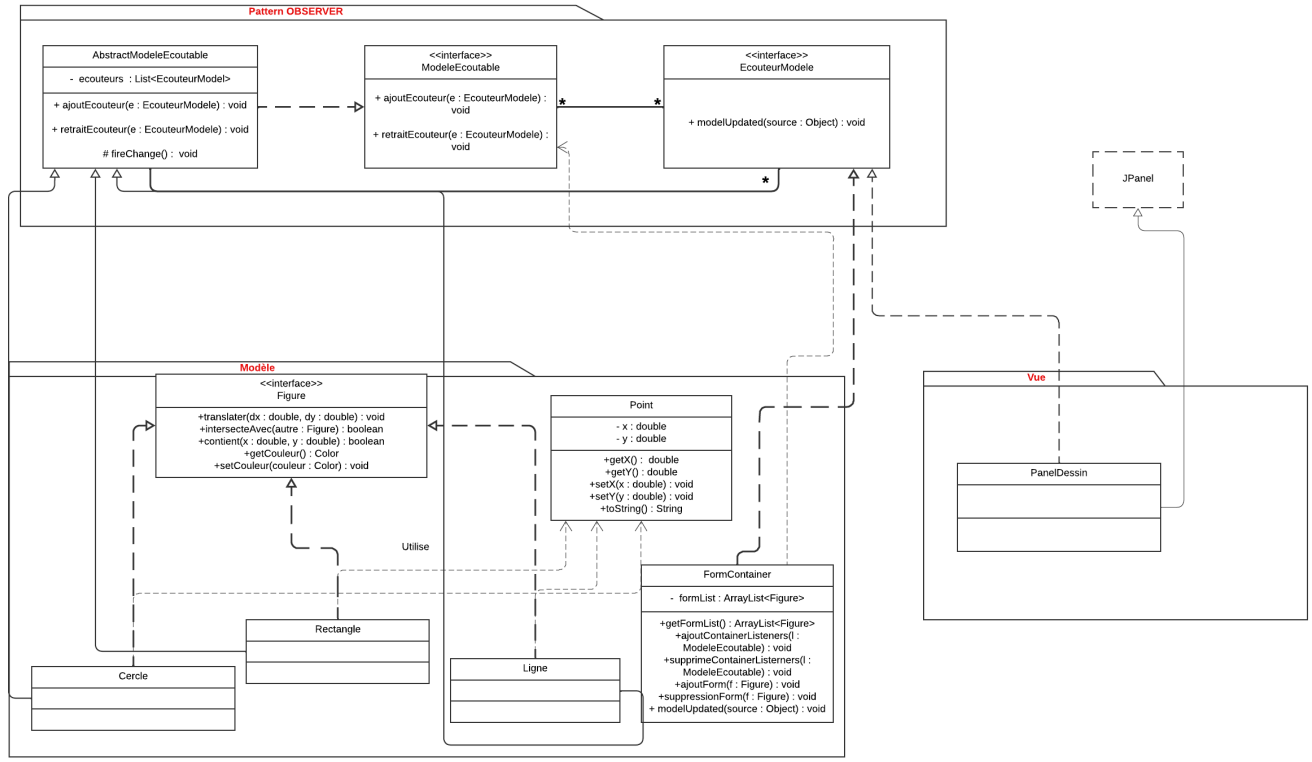


FIGURE 2 – Diagramme du pattern observer

## 4.3 Pattern State

Le pattern **State** est un patron de conception comportemental qui permet de modifier le comportement d'un objet lorsque son état interne change. L'objet donne l'impression qu'il change de classe.
L'idée principale du pattern State est de permettre à notre application de changer de comportement durant l'exécution. Ce pattern ne réagit pas directement sur notre modèle il passe le résultat des calculs au pattern **Commande** qui lui-même les utilise pour réaliser une action.

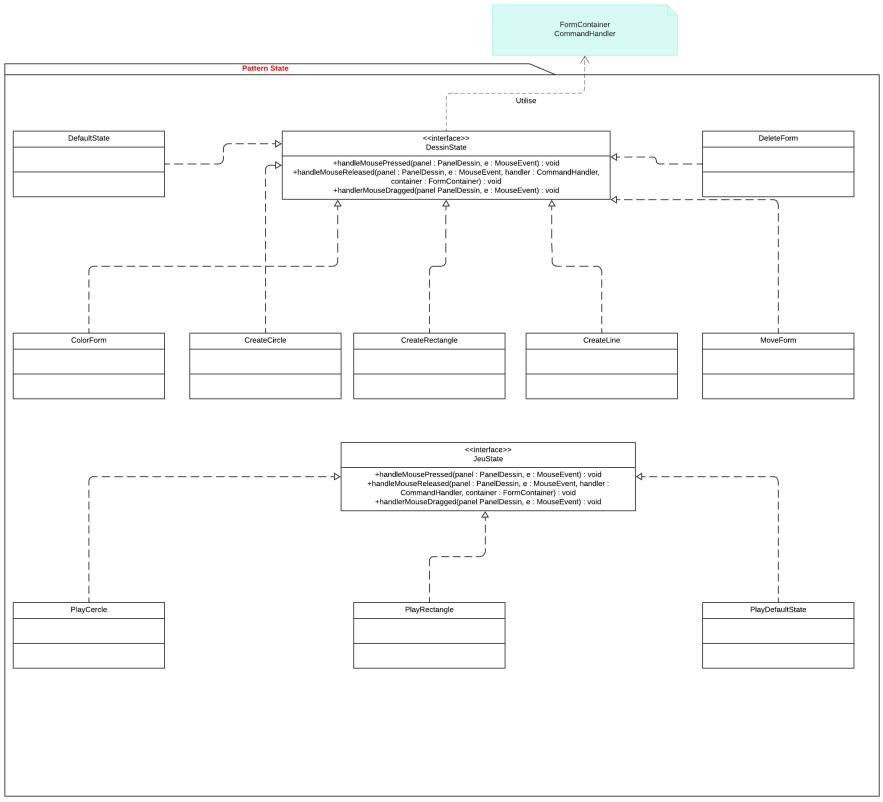


FIGURE 3 – Diagramme du pattern state

- **DefaultState** : cette représente l'état du lancement du jeu ;
- **CreateCircle** : cette classe permet de calculer(le centre + le rayon) pour pouvoir créer le créer un cercle ;
- **CreateReactangle** : cette classe s'occupe des calculs (coin + largeur + longueur) pour pouvoir créer un rectangle.
- **DeleteForm** : cette classe nous permet de faire une vérification pour savoir si c'est un cercle ou un rectangle avant de la supprimer.
- **MoveForm** : cette classe permet de calculer des nouvelles variables pour pouvoir déplacer une forme sélectionnée, un nouveau coin pour un rectangle et un nouveau centre pour un cercle.

La classe state a deux classes principales :

- Une classe pour le mode dessin nommé **DessinState**
- Une classe pour le mode jeu nommé **JeuState**

## 4.4 Pattern Command : Compensation (Undo et Redo)

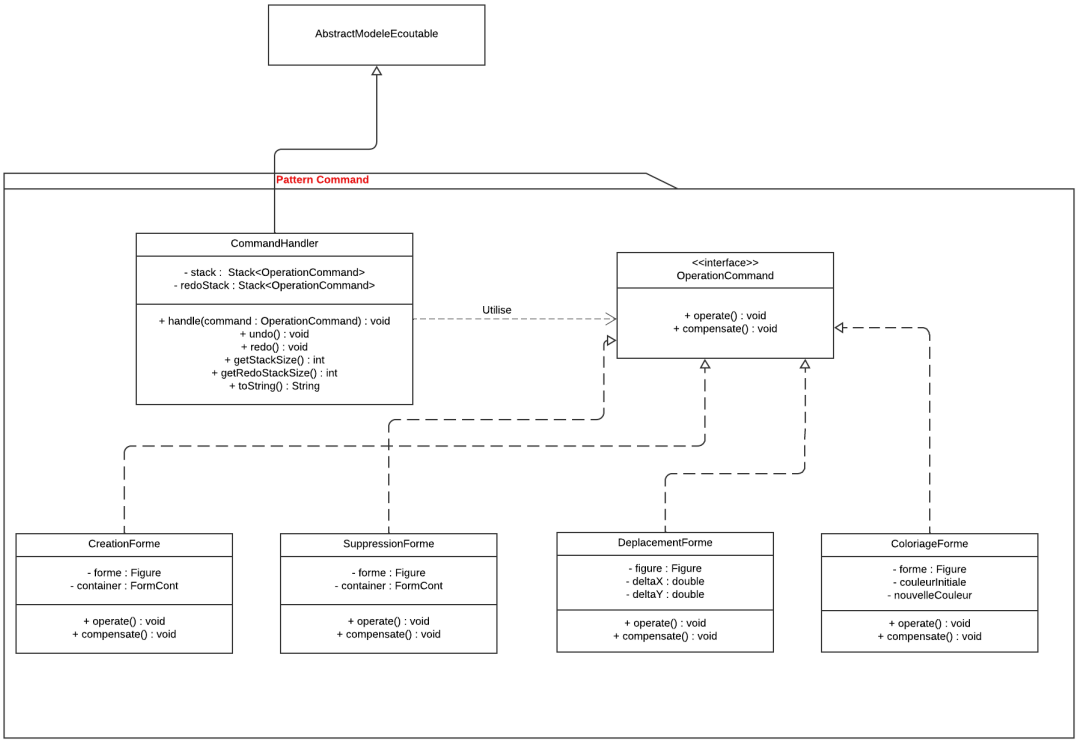


FIGURE 4 – Diagramme du pattern Command

**Command** est un patron de conception comportemental qui prend une action à effectuer et la transforme en un objet autonome qui contient tous les détails de cette action. Cette transformation permet de paramétrer des méthodes avec différentes actions, planifier leur exécution, les mettre dans une file d'attente ou d'annuler des opérations effectuées. Cette séparation du code des actions à celui qui appelle ces actions permet encore une fois d'améliorer la modularité de l'application.
On utilise ce pattern pour implémenter deux fonctionnalités :

- Undo : Cette fonction permet d'annuler la dernière commande exécutée.
- Redo : Cette fonction permet d'exécuter la dernière commande annulée.

Et plus précisément la variation **Compensation**, via les méthodes *operate()* et *compensate()*. Compensation définit en effet pour chaque action, l'action qui la compense. A titre d'exemple, l'action qui compense la création d'une forme est la suppression de la même forme.

## 4.5 Pattern Adapter

Le pattern Adapter est un patron de conception structurel qui permet de faire collaborer des objets ayant des interfaces normalement incompatibles. Nous l'utilisons dans notre application pour convertir notre FormContainer en JTable afin d'afficher des informations sur les formes déposées par le joueur.
Diagramme UML

## 4.6 Factory

Le pattern Factory permet de déléguer la création d'objets à des sous-classes, ce qui améliore l'encapsulation, la flexibilité et l'extensibilité du code. Dans notre logiciel, il est particulièrement utile pour la création de formes. En effet, nous délestons les classes qui implémentent l'interface Figure de la création de celles-ci dans une classe FormesFactory. Celle-ci définit les méthodes permettant de créer des Cercles, des Rectangles et des Points, ainsi qu'une méthode permettant de créer les formes obstacles du mode jeu. Nous y avons intégré une gestion dynamique de la difficulté en augmentant le nombre de formes en fonction du niveau actuel.

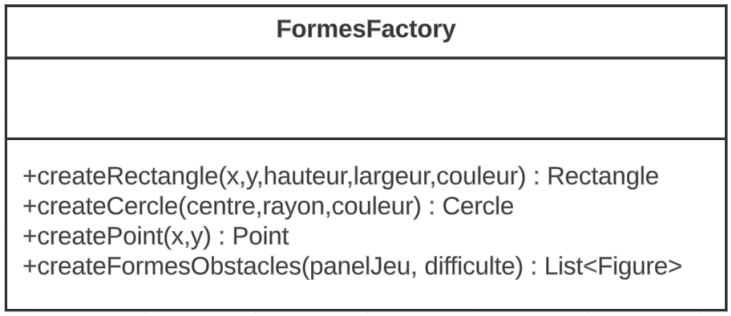


FIGURE 5 – Diagramme du pattern Factory

## 4.7 Strategy

Le pattern Strategy est utilisé pour définir une famille d'algorithmes, encapsuler chacun d'eux et les rendre interchangeables. Dans notre logiciel, nous l'appliquons pour contrôler la difficulté du jeu. Nous avons défini une interface SolverStrategy avec différentes implémentions pour chaque niveau de difficulté : EasyModeSolver et MediumModeSolver. Chaque implémentation définit les règles spécifiques de solver, à savoir le nombre d'itération et la métrique de grossissement. En utilisant le pattern Strategy, nous pouvons facilement changer le comportement de la difficulté du jeu pendant l'exécution en

remplaçant simplement l'objet de stratégie actuel par un autre. Cela permet une gestion dynamique de la difficulté et offre une grande flexibilité pour ajuster le jeu.

## 4.8 Singleton

Le pattern Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance. Dans notre application, on l'utilise pour créer une instance unique de MenuChoix. On a rendu le constructeur de MenuChoix privé et ajouté une méthode statique getInstance() pour s'assurer qu'on ne peut créer qu'une seule instance de MenuChoix. Ça permet d'avoir un seul menu ouvert dans l'appli, évitant les doublons et les confusions.

# 5 Expérimentations et Usages

## 5.1 Utilisation

Nous avons développé notre application avec du java standard Édition 11 pour des raisons de comptabilité avec les ordinateurs de l'université. Elle peut s'exécuter sur les distributions Linux avec interface graphique telle que : Ubuntu, mac Os et windowns. Tous les membres du groupe ont opté pour l'utilisation du logiciel NetBeans afin de favoriser le lancement du projet avec un build ant toujours valide. Il suffit à l'utilisateur d'aller dans les paramètres du projet et de changer la version du JDK pour que cela fonctionne sur sa machine.

## 5.2 Lancement de l'application

Vous devez ouvrir un terminal à l'emplacement du dossier du projet où se trouve le fichier **build.xml**. Pour :

- Lancer l'application, exécuter la commande **ant run**. Une fois la commande **ant run** exécutée, il vous sera demandé de choisir un mode :
  - Mode Dessin
  - Mode Jeu

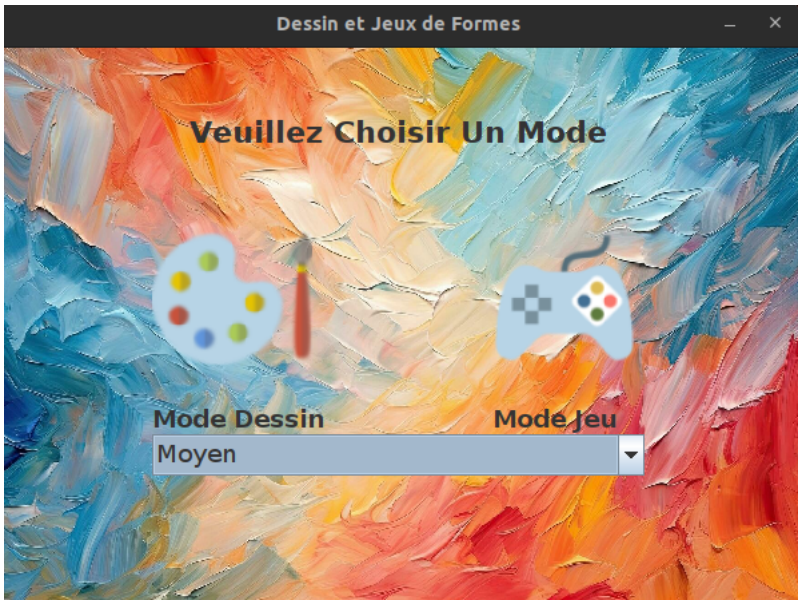


FIGURE 6 – Interface de l'application

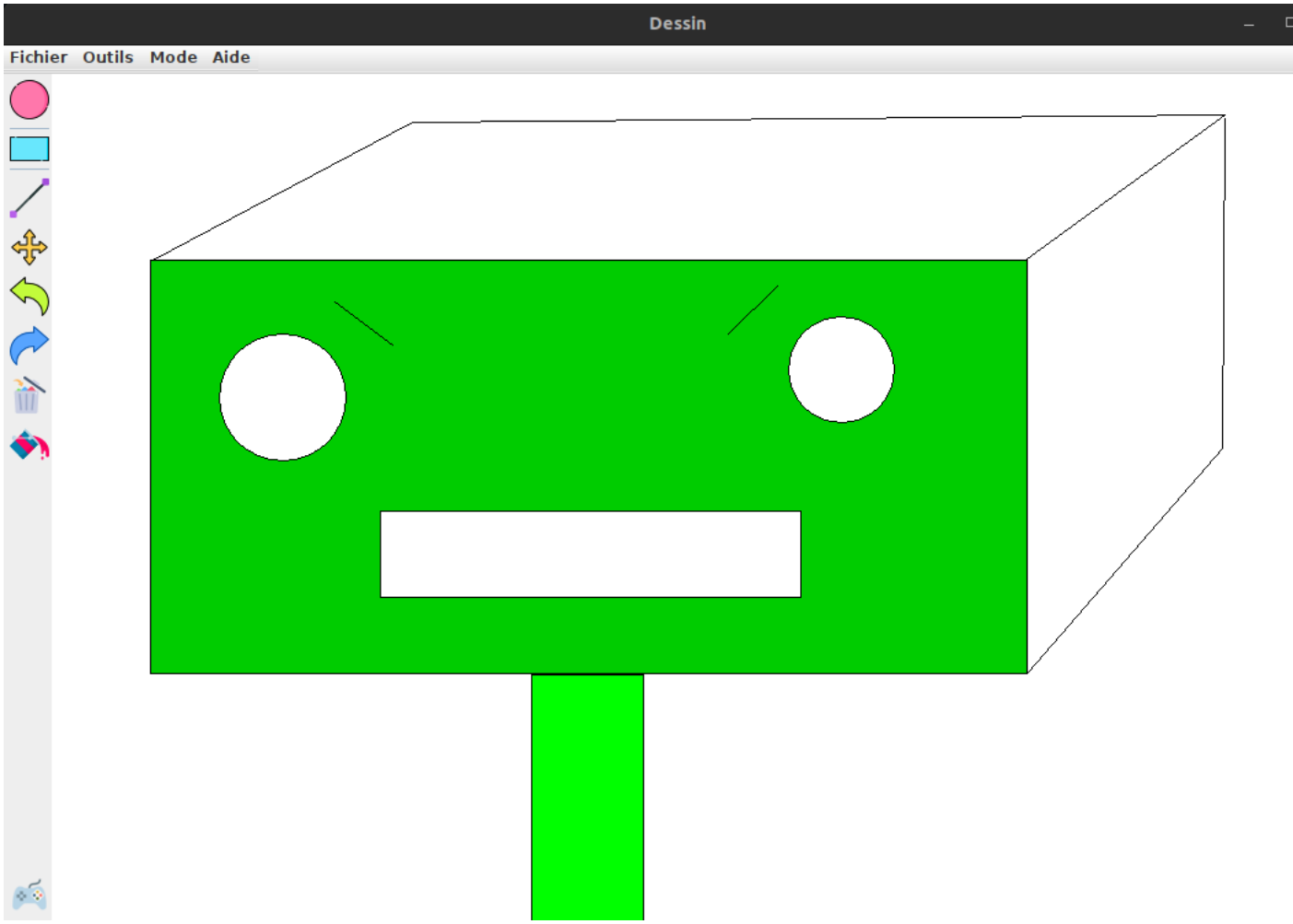


FIGURE 7 – Aperçu mode dessin

- Initialiser le projet : **ant init**
- Compiler le projet : **ant compile**
- Générer le Javadoc : **ant javadoc**
- Nettoyer le projet : **ant clean**
- Lancer le test : **ant test**

Vous pouvez également lancer le projet avec NetBeans en important celui-ci.

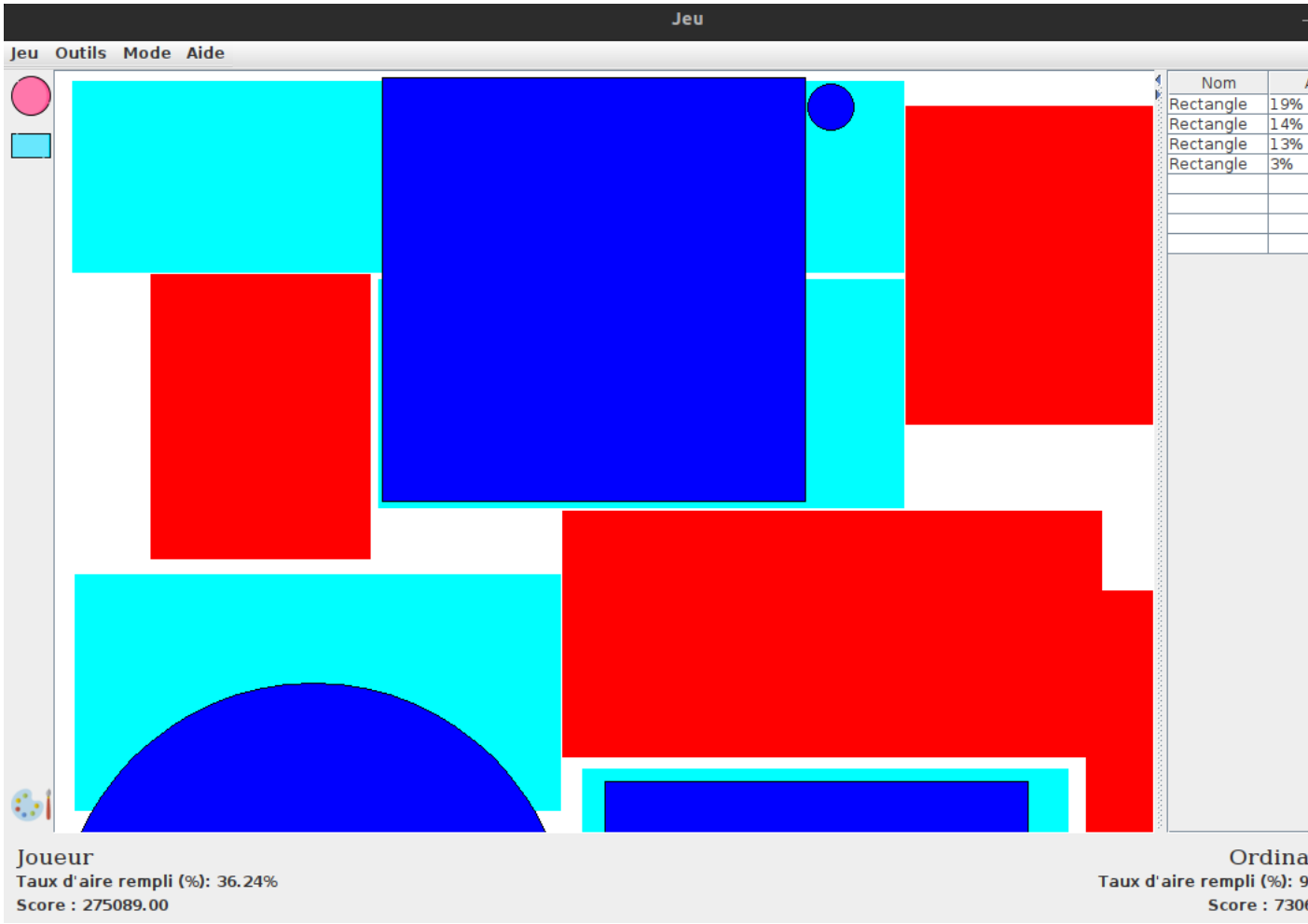


FIGURE 8 – Une fin de partie (Avec bug sur le solver) Cyan = joueur, Bleu foncé = ordinateur

## 5.3 Test de l'application

Nous avons réalisé le test de notre application en utilisant le framework open source Junit pour le développement et l'exécution de tests unitaires. Elle teste l'ensemble des méthodes pertinentes du modèle. Ces tests se sont avérés très utiles au début du projet quand nous modifions souvent le modèle pour y ajouter des éléments, cela nous a permis de ne pas créer des bugs cachés.

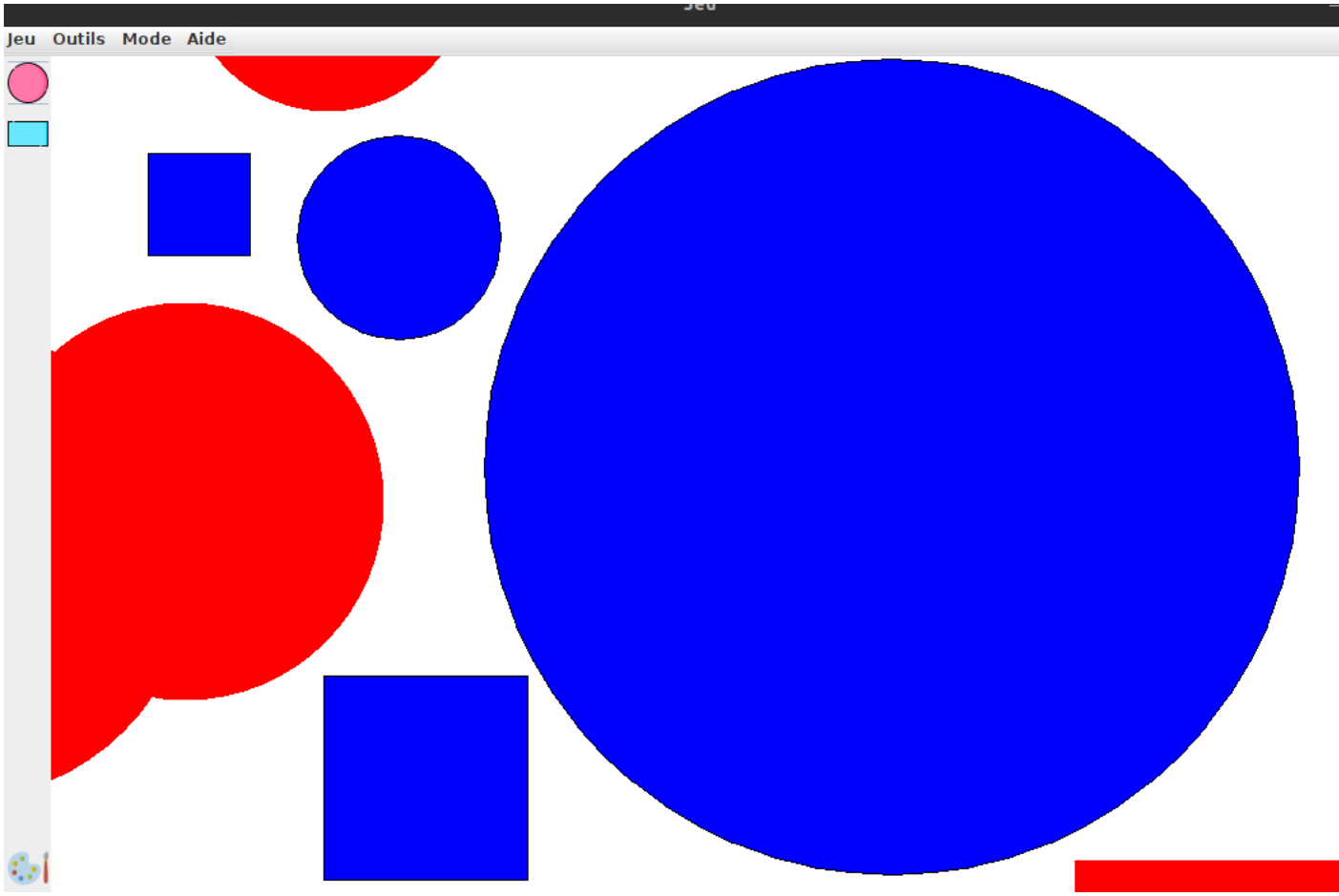


FIGURE 9 – Une solution du solver quand il n'y a avait pas de bug en mode intermédiaire avec iter = 10000000 et métrique de grossissement à 1.1

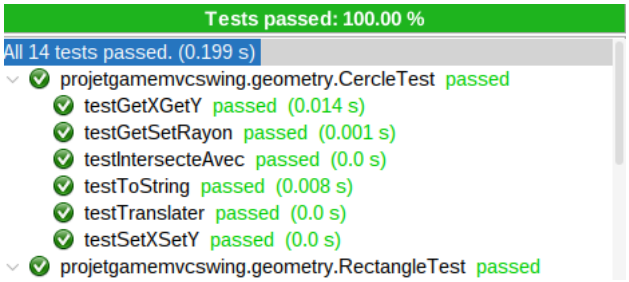


FIGURE 10 – Tests validés avec succès sur NetBeans

# 6 Conclusion

## 6.1 Le projet...

Ce projet est un très bon entraînement pour manipuler les différents patrons de conception que l'on a étudié tout au long du semestre. En effet, au fur et à mesure de l'avancée du développement, nous nous sommes bien vite rendus compte que l'implémentation des

patterns permettaient une grande flexibilité du code et souligne donc leur importance lorsque l'on veut rendre une application modulaire et extensible. De plus, un logiciel de dessin de formes laisse place à une certaine créativité, malgré la rigueur qu'exige le développement d'applications. Cela a su stimuler notre imagination et garder notre motivation intacte du début à la fin.

## 6.2 ...et ses pistes d'amélioration

Après plusieurs dizaines d'heures de travail, nous sommes plutôt satisfaits du rendu que nous proposons. Notre code est scalable et optimisé, il y a très peu de redondance de code grâce aux patterns utilisés, et il restera une nouvelle référence dans nos dépôts github sur les patrons de conception. Notre logiciel est fonctionnel, et nous avons pu mettre en place la partie jeu qui était optionnelle, même si elle possède encore quelques ajustements à faire (bug sur le solveur sur les derniers commits, mauvais calculs d'aires, JTable mal utilisée). Nous avons essayé de dissimuler à l'utilisateur le calcul des solutions pendant que l'utilisateur joue d'où les 10 secondes pour jouer, et nous nous sommes questionnés sur le threading pour ne pas brider les performances. Nous proposons 3 axes de développement à explorer :

- Implémenter davantage de patterns de conception, notamment Composite ou encore Decorator pour l'aspect visuel du logiciel ou Abstract Factory pour le style.
- Ajouter des solveurs, celui actuel étant aléatoire, ce problème d'optimisation est très intéréssant étant de classe NP-Difficile. Plus tard mettre en place le système de plugin pour permettre d'ajouter des algorithmes d'intelligence artificielle plus performants facilement serait très intéréssants pour parfaire ce projet.
- Un système de seed pour permettre de retrouver une génération aléatoire précédente et pourquoi pas itérer 1 million de fois pour trouver la configuration qui donne le plus de mal à notre solveur pour de la recherche.