Université de Caen Normandie

Patrons de conception & Interactions Humain-Machine
Master 1 Intelligence Artificielle et Facteurs Humains

Jeu de mémorisation de formes en Java

Encadré par :
M. Yann Mathet

Réalisé par :
Eldis YMERAJ
Fred MOREL
Remi DE FARIA
Redwan OMARI

Année Universitaire 2024-2025

Table des matières

Table des figures

Introduction

Le but de ce projet est de réaliser une application de jeu de mémorisation de formes en Java avec interface graphique, en respectant l'architecture MVC (Modèle-Vue-Contrôleur) et en intégrant un maximum de patrons de conception étudiés durant le cours.

Les grandes lignes du projet sont les suivantes :

- développer un modèle complètement indépendant de la vue et du contrôleur ;
- concevoir une interface graphique intuitive permettant à l'utilisateur de dessiner des formes (rectangles, cercles) via des interactions souris ;
- proposer plusieurs modes de jeu :
  - un mode solo avec des figures prédéfinies à mémoriser ;
  - un mode deux joueurs où chacun crée et reproduit des ensembles de formes ;
  - un mode aléatoire générant dynamiquement les formes ;
- mettre en place différentes stratégies d'évaluation permettant de comparer la proposition du joueur avec les formes initiales (évaluation par similarité ou par précision) ;
- ajouter des fonctionnalités d'annulation et de rétablissement d'actions (undo-redo) ;
- structurer le code autour de patrons tels que **Strategy**, **Command**, **State**, **Singleton**, et **Observer**, pour garantir une architecture souple, maintenable et extensible.

L'application intègre également une interface composée de boutons permettant de sélectionner le type de forme à dessiner, de valider la reproduction, ou de quitter la partie. Les résultats sont affichés dynamiquement au fur et à mesure des tours de jeu, en fonction du mode choisi.

1 Fonctionnalités du logiciel

1.1 Généralités

Notre application repose sur la manipulation de formes géométriques simples — cercles et rectangles — dans une interface graphique intuitive. L'utilisateur peut dessiner, annuler ou refaire une action, valider une tentative, et obtenir une évaluation sous forme de score. Le logiciel propose trois modes de jeu différents et implémente plusieurs patrons de conception pour rendre le système souple et évolutif.

L'objectif principal étant la mémorisation visuelle, l'application affiche un ensemble de formes pendant un court instant, puis efface la scène et invite l'utilisateur à reproduire les formes de mémoire.

1.2 Dessin libre

Le mode de dessin libre n'est pas directement proposé comme mode autonome, mais fait partie intégrante de l'interface utilisateur dans les différents modes de jeu. L'utilisateur peut, à tout moment après affichage des formes à mémoriser, utiliser la souris pour dessiner librement des cercles ou des rectangles en sélectionnant l'outil correspondant via les boutons disponibles.

Les actions sont totalement réversibles grâce aux boutons `Undo` et `Redo`, mis en œuvre à l'aide du patron *Command*. Le bouton `Valider` permet de soumettre une tentative, qui sera évaluée par l'une des stratégies de notation (précision ou similarité). Enfin, un bouton `Quitter` permet d'interrompre proprement la partie.

1.3 Jeu

Le cœur du logiciel est constitué de trois modes de jeu :

- **Mode Solo (formes enregistrées)** : l'utilisateur doit mémoriser et reproduire des figures prédéfinies. Chaque étape présente un nouveau motif plus complexe.
- **Mode Deux Joueurs** : chaque joueur crée à tour de rôle un ensemble de quatre formes que l'autre doit mémoriser puis reproduire. L'alternance des rôles permet une compétition sur 10 tours, avec un score cumulé calculé à la fin.
- **Mode Aléatoire** : des formes sont générées aléatoirement à l'écran pour un temps limité. Le joueur doit les reproduire ensuite, et plus il réussit, plus le nombre de formes augmente au niveau suivant.

Pour chaque mode, une stratégie d'évaluation est sélectionnée dès le lancement de l'application (à l'aide du patron *Strategy*). Le score final est affiché soit sous forme d'un message individuel (mode solo et aléatoire), soit comparativement (mode deux joueurs).

L'interface contient aussi une zone dédiée à l'affichage des scores au fur et à mesure des tours, permettant un retour immédiat sur la performance.

1.4 Diagramme UML

Nous avons alors décidé d'implanter ce jeu avec les classes et les patterns suivants :

Pour une meilleure lisibilité des classes et des patterns, n'hésitez pas à voir le fichier *Diagramme Classe Projet Memorisation Patrons* à la racine du dépôt git.

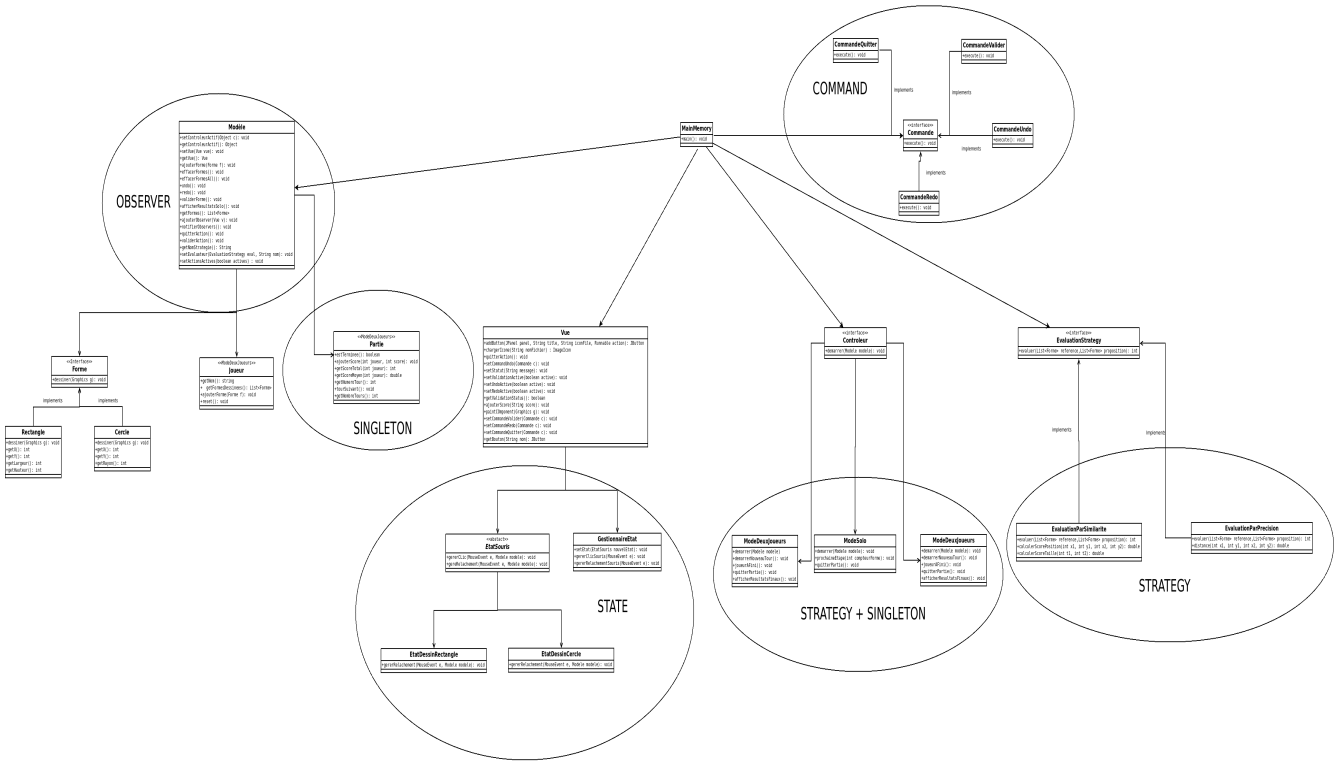

Figure 1 – Diagramme de classes du projet de mémorisation

2 Organisation du projet

2.1 Répartition des tâches

L'ossature du projet s'est faite par le groupe entier. Notre volonté première était de s'accorder sur le choix d'évaluation des formes par rapport à celles affichées. Nous avons alors décidé de partir sur un modèle assez intuitif :

- Pour les formes rectangulaires : On regarde la distance du coin supérieur gauche entre les deux formes et on calcule un premier score. Dans un second temps, on regarde la différence de périmètre entre les deux formes et on calcule un second score.
- Pour les formes circulaires : On regarde la distance entre les deux centres des cercles et on calcule un premier score. Le second score est calculé selon la différence de rayon entre les deux formes.

Après plusieurs tests effectués (voir subsection 4.3), on a réalisé que cette méthode d'évaluation avait tout de même plusieurs défauts. Nous avons alors corrigé cela dans la partie utilisation (voir subsection 4.1).

Ensuite, et pour respecter les consignes du projet, celui-ci a été réparti tel que :

- La partie "classique" (ModeSolo), où dix dessins sont pré-enregistrés et proposés au joueur a été faite par *Fred MOREL*. Après chacun des dessins, un score est donné selon l'évaluateur énoncé ci-dessus. Le score final du joueur est alors la moyenne de ses dix scores.
- La partie "Random" (ModeRandom), où, de la même manière que le mode classique, dix dessins apparaissent mais de cette fois avec des formes de taille aléatoire. Dans cette partie, le nombre de formes par dessin est proportionnel au niveau pour s'ajuster à la difficulté. Comme pour la première partie, le score final du joueur sera la moyenne de ces dix scores. Cette partie a été réalisée par *Rémi DE FARIA*.
- Finalement, la partie a deux joueurs "ModeDeuxJoueurs", a été réalisée par *Eldis YMERAJ et Redwan OMARI*. Cette partie, étant plus complexe que les deux premières car elle doit gérer les intéractions

entre les deux joueurs, a une architecture légèrement différente des deux premières parties. Le principe global de ce mode est de proposer à deux joueurs, chacun leur tour, de dessiner quatre formes et de les proposer à l'autre joueur. Chacun des joueurs aura alors à faire cinq dessins. Après chaque dessin proposé par le joueur devant recréer les formes mémorisées, un score lui sera donné. Le joueur qui gagne est celui ayant la meilleure moyenne de ses cinq scores.

Pour plus d'informations sur le jeu et les différentes caractéristiques de ses modes, voir la partie sur le jeu (subsection 1.3).

La réalisation du diagramme UML implantant les différentes classes et les différents patterns utilisés a été réalisé par *Fred MOREL*.

L'implantation des différents patterns au sein des classes a été faite par *Eldis YMERAJ*. (voir la section sur les patterns section 3).

2.2 Architecture du projet

L'architecture de notre projet suit fidèlement le modèle MVC (Modèle-Vue-Contrôleur), demandé dans le sujet. Cela permet une séparation claire entre la logique métier, l'interface graphique et la gestion des événements.

Nous avons structuré le projet autour de trois packages principaux :

- **modele** : contient toutes les classes représentant les entités manipulées (formes, partie, joueur, modèle central). Ce package est totalement indépendant des interfaces graphiques.
- **controleur** : regroupe les différentes logiques métier liées aux modes de jeu (solo, aléatoire, deux joueurs), ainsi que les commandes utilisateurs (`CommandeUndo`, `CommandeRedo`, etc.), implémentant des patterns comme *Strategy*, *Command* et *Singleton*.
- **vue** : gère l'affichage et les interactions souris. On y trouve les classes `Vue` (interface principale) ainsi que les états de souris dans le sous-package `vue.state`, suivant le pattern *State*.

Chaque couche respecte son rôle sans dépendances directes entre Vue et Modele : seul le contrôleur agit comme médiateur. Cela garantit la maintenabilité et l'extensibilité de notre application.

3 Les patterns utilisés

Pour ce projet, nous avons implémenté plusieurs patrons de conception parmi ceux étudiés en cours. Ces patterns nous ont permis de structurer le projet de façon claire, modulaire et extensible, tout en respectant les principes de responsabilité unique, d'ouverture/fermeture et d'indépendance entre les composants. Voici les principaux patterns utilisés :

3.1 MVC

L'architecture générale de notre application repose sur le pattern **Modèle-Vue-Contrôleur** (MVC), qui permet une séparation claire entre les données, leur affichage et la logique métier.

- Le **Modèle** (`Modele`, `Partie`, `Forme`, etc.) contient les données principales, comme les formes dessinées, les scores, ou encore les joueurs. Il est totalement indépendant de la vue et du contrôleur.
- La **Vue** (`Vue`) s'occupe de tout l'affichage graphique, ainsi que des interactions de l'utilisateur (clics souris, boutons, etc.). Elle est automatiquement mise à jour dès qu'un changement est détecté dans le modèle.
- Le **Contrôleur** (interfaces comme `Controleur`, implémentées par `ModeSolo`, `ModeDeuxJoueurs`, `ModeRandom`) orchestre les interactions entre la Vue et le Modèle.

Cette organisation nous a permis de rendre le code clair, facilement testable, et surtout extensible à de nouveaux modes de jeu ou de nouvelles fonctionnalités.

3.2 Observer

Pour respecter le découplage entre la Vue et le Modèle, nous avons appliqué le pattern **Observer**. Il permet à la Vue d'être automatiquement informée lorsqu'un changement est effectué dans le Modèle.

La classe `Modele` conserve une liste d'observateurs, et chaque fois qu'une action (ajout de forme, undo/-redo, etc.) est réalisée, elle appelle la méthode `notifierObservers()`. La Vue, inscrite comme observateur, réagit alors en redessinant la zone graphique grâce à `repaint()`.

Ce mécanisme assure une cohérence en temps réel entre les données internes et leur représentation graphique, tout en maintenant une parfaite indépendance entre les deux composants.

3.3 State

Pour gérer les différents comportements de la souris (dessin d'un cercle, d'un rectangle, redimensionnement, suppression...), nous avons utilisé le pattern **State**. Il évite d'avoir une logique conditionnelle complexe dans une seule méthode et permet à chaque état de gérer indépendamment les événements souris.

Chaque état (comme `EtatDessinCercle`, `EtatDessinRectangle`) hérite d'une classe abstraite `EtatSouris`, et redéfinit les comportements à adopter lors des clics, déplacements et relâchements de souris.

Le changement d'état est centralisé dans la classe `GestionnaireEtat`, ce qui nous a permis d'ajouter ou de modifier des fonctionnalités sans toucher aux autres états.

3.4 Command

Pour implémenter des actions utilisateur annulables comme `undo`, `redo`, `valider`, ou `quitter`, nous avons utilisé le pattern **Command**.

Chaque commande implémente l'interface `Commande` avec une méthode `execute()`. La Vue agit comme invocateur : elle appelle simplement la commande associée lors d'un clic sur un bouton. Par exemple, `CommandeUndo`, `CommandeRedo`, `CommandeValider` sont des classes concrètes qui encapsulent une action métier à exécuter.

Ce pattern rend l'application modulaire : on peut ajouter de nouvelles commandes facilement, sans modifier la Vue ni la logique du Modèle.

3.5 Strategy

Le pattern **Strategy** est utilisé pour permettre le choix dynamique d'un mode d'évaluation. L'interface `EvaluationStrategy` est implémentée par `EvaluationParSimilarite` et `EvaluationParPrecision`, qui proposent deux méthodes de calcul de score différentes.

Le `Modele` contient un champ de type `EvaluationStrategy`, et le contrôleur injecte la bonne stratégie selon le mode de jeu actif. Ce découplage permet d'enrichir ou de remplacer les évaluateurs sans modifier la logique de base du programme.

3.6 Singleton

Nous avons utilisé le pattern **Singleton** pour les différents modes de jeu. Chacun de ces modes ne peut alors avoir qu'au plus une instance de son mode. Nous avons ajouté le pattern Singleton à la classe `Partie`. Étant donné que la partie en cours ne doit exister qu'en une seule instance (elle contient les scores, les tours, les joueurs...). Pour toutes ces classes et pour rester conforme au pattern Singleton, nous avons implémenté une méthode `getInstance()` pour garantir leur unicité.

Cela permet d'accéder aux différentes classes de façon globale tout en évitant d'en créer plusieurs instances accidentellement. C'est particulièrement utile dans le mode deux joueurs où cette instance centrale doit être partagée entre différentes phases du jeu.

4 Expérimentations et Utilisation

4.1 Utilisation

Le projet est conçu pour être exécuté facilement sur n'importe quel système (Linux, macOS ou Windows), à condition d'avoir Java et Apache Ant installés. Tout est automatisé via un fichier `build.xml` qui compile le projet, génère la Javadoc, crée un exécutable et permet de lancer l'application.

Prérequis

- **Java 8 ou supérieur**
  Le projet est compilé avec un encodage UTF-8 sans cibler une version spécifique dans le `build.xml`, ce qui le rend compatible avec toute version récente de Java. On recommande Java 8 minimum. Pour vérifier la version :
  `java -version`

- **Apache Ant**
  L'outil Ant permet de compiler et exécuter facilement le projet. Il est utilisé à travers le fichier `build.xml`. Pour vérifier l'installation :
  `ant -version`

Récupération et lancement du projet

1. Clonez le dépôt Git :

   ```
   git clone https://git.unicaen.fr/morel246/projet-patrons-dessins.git
   cd projet-patrons-dessins
   ```

2. Nettoyez les éventuelles anciennes compilations :

   ```
   ant clean
   ```

3. Compilez le projet, générez la documentation et créez l'exécutable :

   ```
   ant dist
   ```

4. Lancez directement l'application :

   ```
   ant run
   ```

Structure générée

La commande `ant dist` effectue les opérations suivantes :
- Compilation dans le dossier `build/`
- Création du JAR `jeu-memoire.jar` dans `dist/`
- Génération de la documentation Javadoc dans `doc/`
- Copie automatique des fichiers images (`.png`) dans `dist/vue/`

Remarques complémentaires

Le projet est conçu pour fonctionner sans dépendances externes. Il peut être ouvert dans n'importe quel IDE ou lancé en ligne de commande avec Java. En cas de problème d'encodage sur certains systèmes, l'attribut `encoding="UTF-8"` dans le `build.xml` permet de garantir la bonne lecture des fichiers source.

4.2 Lancement de l'application

Une fois le projet compilé avec la commande `ant dist`, il est possible de lancer directement l'application à l'aide de :

```
ant run
```

Cela exécute le fichier `jeu-memoire.jar` qui se trouve dans le dossier `dist/`. L'application démarre alors en affichant deux fenêtres successives pour configurer la partie avant de commencer à jouer.

Choix du mode de jeu La première fenêtre demande à l'utilisateur de choisir le mode de jeu souhaité. Trois options sont disponibles :

- **Solo – Formes enregistrées** : le joueur doit mémoriser et reproduire des ensembles de formes dessinées à l'avance (ex : bonhomme de neige, maison...).
- **Deux Joueurs** : chaque joueur dessine quatre formes que l'autre devra ensuite reproduire de mémoire. Les rôles s'alternent automatiquement pendant la partie.
- **Solo – Formes aléatoires** : les formes sont générées aléatoirement à chaque niveau, avec une difficulté croissante.

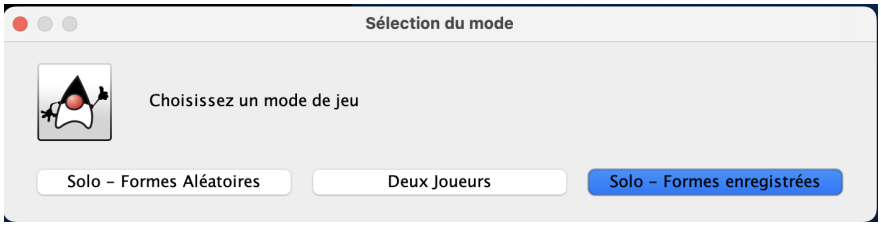

Figure 2 – Fenêtre de sélection du mode de jeu

Choix de la stratégie d'évaluation Une fois le mode choisi, une seconde fenêtre s'affiche pour permettre à l'utilisateur de sélectionner la méthode d'évaluation utilisée pour comparer les formes dessinées à celles attendues :

- **Évaluation par Précision** : méthode plus stricte, avec seuils sur la position et la taille. Deux formes proches mais pas identiques donneront un score plus faible.
- **Évaluation par Similarité** : méthode plus tolérante, prenant en compte le type, la position et la taille avec une pondération plus souple.

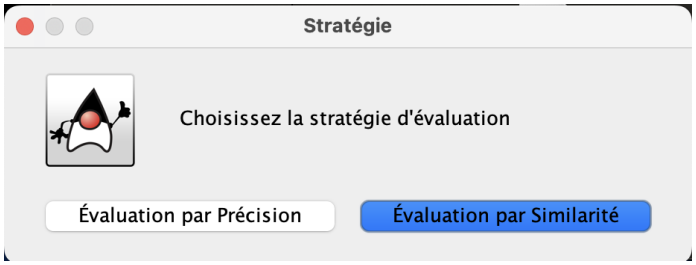

Figure 3 – Fenêtre de sélection de la stratégie d'évaluation

Début de la partie Une fois les deux choix effectués, l'interface principale du jeu s'ouvre. L'utilisateur accède à une barre de boutons (forme à dessiner, undo, redo, valider, quitter), une zone de dessin, et un encadré affichant les scores obtenus.

Le reste de l'expérience dépend du mode sélectionné. Le fonctionnement est fluide et ne nécessite pas d'étapes supplémentaires : le jeu se lance immédiatement et guide le joueur via des messages affichés à l'écran selon les phases (dessin, reproduction, fin de partie...).

4.3 Tests

Une fois toutes les fonctionnalités implémentées, nous avons pris le temps de tester chaque mode de jeu et chaque stratégie d'évaluation pour vérifier que l'application se comportait comme prévu et que l'expérience utilisateur restait agréable.

Évaluation par similarité Cette stratégie est conçue pour être plus tolérante : même si les formes ne sont pas parfaitement identiques, le joueur peut quand même obtenir un score satisfaisant. Cela permet de ne pas trop pénaliser les petites erreurs de placement ou de taille.

Lors de nos tests, on a par exemple tenté de reproduire un dessin simple composé de deux cercles et d'un rectangle, dans le mode deux joueurs. Le score obtenu était de 83/100, ce qui nous a paru juste : le dessin était globalement fidèle à l'original, malgré un léger décalage du rectangle.

La stratégie repose sur plusieurs critères :

— le type de forme (Cercle ou Rectangle),

— la distance entre les centres des formes,

— les différences de taille (écart de rayon, largeur ou hauteur).

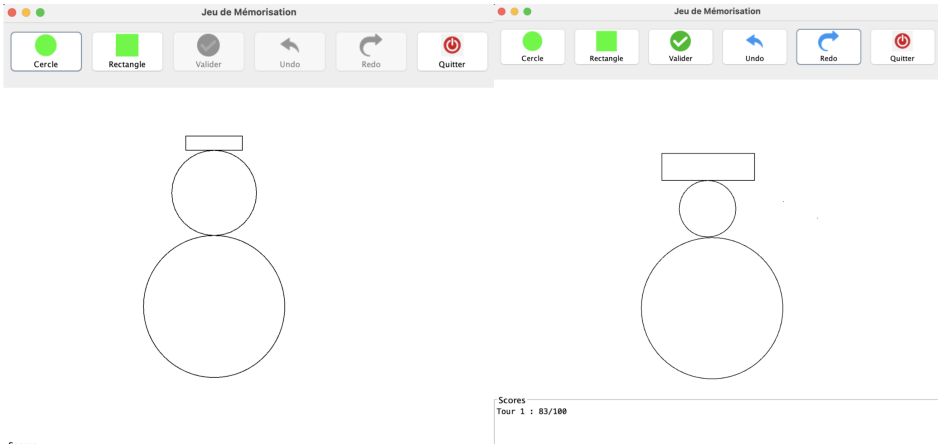

Figure 4 – Exemple de test avec évaluation par similarité (83/100)

Évaluation par précision Cette stratégie est beaucoup moins tolérante que la précédente : dès qu'une forme est un peu trop décalée ou que sa taille ne correspond pas presque parfaitement, le score chute rapidement. C'est idéal pour ceux qui veulent s'imposer un vrai challenge de précision.

Lors de nos tests, on a repris le même dessin que pour la similarité (deux cercles et un rectangle), mais cette fois évalué avec la stratégie par précision. Résultat : 66/100. Pourtant le dessin était pratiquement le même. Cette différence de score reflète bien la rigueur de cette stratégie, qui laisse peu de place à l'approximation.

Voici les critères principaux :

— le type de forme doit être identique (pas de cercle à la place d'un rectangle),

— la position doit être très proche (écart de moins de 40 pixels),
— la taille doit quasiment correspondre (moins de 15 pixels de différence).

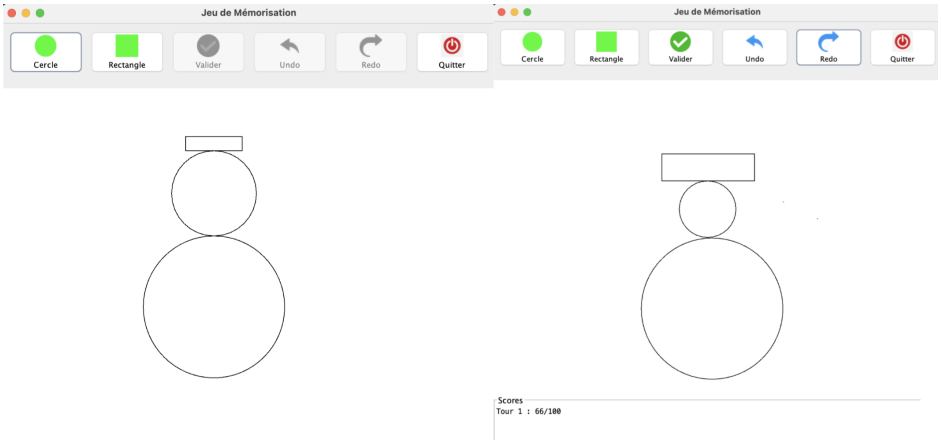

Figure 5 – Exemple de test avec évaluation par précision (66/100)

Choix dynamique de la stratégie Ce qui a été pertinent pendant les tests, c'est le fait de pouvoir choisir la stratégie juste avant de commencer la partie. Grâce à l'implémentation du patron *Strategy*, le modèle applique automatiquement la bonne méthode de calcul selon le choix de l'utilisateur, sans que l'on ait à modifier un quelconque paramètre dans le reste du code. C'est simple d'utilisation et très souple du côté du code.

Interface et comportements dynamiques En testant l'application, on s'est rapidement rendu compte que l'interface réagit de manière cohérente et adaptée aux différentes phases du jeu. Les boutons d'action comme `Valider`, `Undo` ou `Redo` s'activent et se désactivent en fonction du contexte, ce qui évite toute confusion pour l'utilisateur. Ce comportement contextuel a vraiment fluidifié l'expérience : on ne s'est jamais retrouvé à chercher quoi faire, tout est déclenché au bon moment.

Les scores sont affichés instantanément après chaque validation, ce qui permet au joueur de suivre sa progression sans avoir besoin d'interactions supplémentaires. L'enchaînement des étapes se fait naturellement, et le dessin reste fluide à la souris, même lorsqu'on change rapidement de forme ou de mode. Ce sont des détails discrets, mais qui font une vraie différence quand on utilise l'application sur plusieurs tours de jeu.

Bilan des tests Les tests se sont bien déroulés dans l'ensemble. Tous les modes de jeu ont été testés plusieurs fois, avec les deux stratégies d'évaluation. Aucun plantage, pas de bug gênant, et surtout une expérience fluide même en enchaînant plusieurs parties. Cela nous a permis de valider la stabilité du jeu, mais aussi de confirmer que la logique d'évaluation donnait des résultats cohérents avec ce que le joueur a réellement dessiné.

Conclusion

Le Projet

Bien qu'ayant travaillé tous les quatre sur le projet durant les séances de TP, il a été nécessaire de beaucoup communiquer et de s'accorder sur les différentes tâches à faire. L'utilisation du dépôt git dès le début du projet nous a été primordiale pour ajouter au fur et à mesure le code adéquat.

De manière générale, nous sommes très satisfaits du rendu final qui répond à tous les critères donnés par le sujet :

1. L'implantation de trois modes de jeu : Avec des dessins pré-enregistrés, avec des dessins aléatoires et avec un mode à deux joueurs. Chacun de ces modes est fonctionnel et répond au cahier des charges fourni.
2. L'implantation de différents patterns vus pendant les séances de TP. Comme le montre la partie sur les patterns (voir section 3), nous avons utilisé un grand nombre de patterns enseignés ce semestre.
3. Facilité de compréhension, maintenance et utilisation : Grâce à l'ajout de ces patterns et à la factorisation de notre code, celui-ci s'inscrit alors pleinement dans ces caractéristiques.

Pistes d'amélioration

Certaines améliorations mineures restent cependant à notifier. Bien que nous ayons implanté plusieurs méthodes de "scoring", le résultat des évaluateurs ne nous plaît toujours pas complètement. Une analyse plus profonde serait nécessaire pour avoir le scoring le plus juste possible.

De plus, malgré que nous ayons fait des efforts sur ce sujet, nous ne sommes pas entièrement satisfaits de l'esthétisme du jeu. La fenêtre de jeu ainsi que ses différents pop-up pourraient être améliorés pour rendre l'expérience utilisateur plus agréable.

Finalement, il serait intéressant d'introduire d'autres formes que celles rectangulaires et circulaires au jeu. Nous nous sommes contentés de ne garder que ces deux formes pour faciliter le calcul du score et ainsi implanter nos idées d'évaluation. L'ajout d'autres formes aurait nécessité une autre approche d'évaluation, domaine dans lequel nous avons choisi de ne pas aller.