Letoublon Thomas

Roustan Liam

Moyaerts Mathieu

**Rapport BDD**

**a) Analyse et justification des scénarios**

**i) Identification des cas de test**

**Démineur :**

Nous avons identifié plusieurs catégories de cas de test pour couvrir le comportement du Démineur :

* **Cas nominaux** : interactions classiques du joueur avec le jeu. Exemples :
  + Cliquer sur une case vide.
  + Poser ou retirer un drapeau.
  + Révéler toutes les cases non piégées.
* **Cas limites** : comportement en bordure de la grille ou avec des données particulières.
  + Cliquer sur une case au coin ou au bord.
  + Premier clic au centre ou à la périphérie.
* **Cas d'erreur / protection** :
  + Cliquer sur une case avec un drapeau.
  + Poser un drapeau sur une case déjà révélée.
  + Révélation de toutes les bombes en cas de défaite.

Notre stratégie vise une **couverture fonctionnelle totale**, en validant les règles de base, les limites, et la résilience aux mauvaises interactions.

**ii) Priorisation des scénarios**

**Démineur :**

L'ordre de développement a suivi une logique de dépendance fonctionnelle :

1. **Scénarios critiques / fondamentaux** :
   * Initialisation de la grille.
   * Premier clic sans bombe.
   * Placement des bombes.
   * Révélation de case.
2. **Scénarios interactifs** :
   * Poser / retirer un drapeau.
   * Empêcher une action sur case marquée.
3. **Scénarios de fin de partie** :
   * Cliquer sur une bombe.
   * Gagner la partie.

Cette priorité garantit une progression stable et testable du moteur du jeu.

**b) Architecture et représentation des données**

**i) Lisibilité des données de test**

Nous avons utilisé différentes structures dans les fichiers. feature :

* **Scenario Outline + Examples** : pour tester les cas paramétrés (dimensions de la grille, valeurs de numéro).
* **Background** : pour préparer un contexte commun (état de la partie en cours, grille initialisée).
* **Scénarios simples** : quand un cas est spécifique et autonome.

Ce choix améliore la lisibilité et réduit la duplication des contextes.

**ii) Extensibilité**

Notre architecture BDD et code est conçue pour être facilement étendue :

* Les StepDefinitions sont modulaires et réutilisables.
* Les Case, Grille, et GameManager sont isolés dans une bibliothèque (Game) utilisable par d'autres jeux si besoin.
* Il est facile d'ajouter d'autres jeux (ex. : Puissance 4, Sudoku) dans des dossiers Features/NouveauJeu/, avec leurs propres steps.
* Les règles peuvent être modifiées en adaptant seulement les méthodes liées dans la classe Grille.

**c) Stragégie BDD et bonnes pratiques**

**i) Langage ubiquitaire**

Nous avons utilisé un **vocabulaire métier** cohérent avec le jeu du Démineur :

* "case", "grille", "bombe", "drapeau", "révéler", "premier clic", "numéro", etc.

Chaque scénario est compréhensible par un utilisateur non technique.

**ii) Réutilisabilité des StepDefinitions**

Nous avons créé :

* des Given génériques ("une partie est en cours", "une case est marquée d'un drapeau")
* des Then réutilisables ("la case est révélée", "la partie est gagnée")

Et lorsque nécessaire, des Steps spécifiques pour les comportements propres ("toutes les bombes sont révélées").

**iii) Maintenance**

* **Structure en dossiers** : les fichiers .feature sont organisés par fonctionnalité (Initialisation, Drapeaux, Fin de partie, etc.)
* **Nommage clair** : des noms explicites pour chaque scénario et step.
* **Modularité** : chaque fonctionnalité a son propre fichier StepDefinitions.cs, ce qui facilite la lecture et la maintenance.

Cette organisation permet d'ajouter ou modifier les scénarios sans impact sur les autres modules du jeu.

**Juste Prix**

**a) Analyse et justification des scénarios**

**i) Identification des cas de test**

**Cas nominaux**

* Proposer un prix dans l’intervalle [minPrix, maxPrix]
* Recevoir la réponse « trop bas », « trop haut » ou « exact » selon la proposition
* Gagner la partie quand la proposition est exacte

**Cas limites**

* Proposer exactement minPrix ou maxPrix
* Atteindre maxEssais essais sans trouver le prix
* Réinitialiser une partie déjà terminée

**Cas d’erreur / protection**

* Dépasser le nombre d’essais autorisés → lever TooManyAttemptsException
* Proposer après la fin de la partie → lever InvalidOperationException
* Proposer une valeur hors de l’intervalle → lever ArgumentOutOfRangeException

**ii) Priorisation des scénarios**

1. **Fondamentaux**
   * Initialisation d’une nouvelle partie (min, max, maxEssais)
   * Génération ou définition du prix secret
2. **Interaction de base**
   * Proposer un prix valide et obtenir le feedback
   * Vérifier que la partie n’est pas terminée après un essai incorrect
3. **Fin de partie**
   * Proposition exacte → partie gagnée
   * Dépassement du nombre d’essais → exception
   * Tentative après fin de partie → exception
4. **Fonctionnalités auxiliaires**
   * Réinitialisation de la partie remet tout à zéro et génère un nouveau secret

**b) Architecture et représentation des données**

**i) Lisibilité des données de test**

* **Background** pour créer et paramétrer la partie (minPrix, maxPrix, maxEssais)
* **Scenario Outline + Examples** pour balayer plusieurs combinaisons de propositions et de feedback
* **Scénarios unitaires** pour les exceptions (TooManyAttemptsException, InvalidOperationException)

**ii) Extensibilité**

* Les StepDefinitions (Given une nouvelle partie, When je propose, Then la réponse est, Then une exception est levée) sont **génériques** et réutilisables.
* La logique de jeu (classe JustePrixGame) est isolée dans la DLL, indépendante de l’UI ou des tests.
* On peut facilement ajouter d’autres variantes (par exemple un mode “avec aide” ou “mode multijoueur”) en créant de nouveaux .feature et de nouveaux Steps sans toucher au cœur du moteur.

**c) Stratégie BDD et bonnes pratiques**

**i) Langage ubiquitaire**

Termes métier clairs : **prix secret**, **essai**, **proposition**, **trop bas / trop haut / exact**, **partie terminée**, **réinitialiser**.

**ii) Réutilisabilité des StepDefinitions**

* **Given** : une nouvelle partie est lancée, le prix secret est X, j’ai déjà fait N essais
* **When** : je propose Y
* **Then** : la réponse est Z, une exception E est levée, la partie est (non) terminée, le nombre d’essais est égal à N

**iii) Maintenance**

* Organisation des .feature par couche : Initialisation, Propositions, Exceptions, Réinitialisation.
* Nommage explicite des scénarios et des Steps.
* Chaque dossier de StepsDefinitions gère un aspect unique du jeu, facilitant la mise à jour ou l’extension.

**Mastermind**

**a) Analyse et justification des scénarios**

**i) Identification des cas de test**

**Cas nominaux**

* Proposer un code de la bonne longueur avec des couleurs valides
* Obtenir le feedback sous forme de *pions noirs* (bonne couleur, bonne position) et *pions blancs* (bonne couleur, mauvaise position)
* Gagner la partie en trouvant exactement le code

**Cas limites**

* Code avec répétitions de couleurs
* Proposition contenant plusieurs fois la même couleur
* Maximum d’essais atteint sans trouver le code

**Cas d’erreur / protection**

* Proposer un code de longueur différente → ArgumentException
* Proposer une couleur hors de la palette autorisée → ArgumentException
* Proposer après la fin de la partie → InvalidOperationException

**ii) Priorisation des scénarios**

1. **Fondamentaux**
   * Initialisation d’une partie (longueur du code, palette de couleurs, maxEssais)
   * Définition ou génération du code secret
2. **Interaction de base**
   * Proposition valide et calcul du feedback
   * Vérifier l’état de la partie après chaque essai
3. **Fin de partie**
   * Proposition exacte → partie gagnée
   * Dépassement du nombre d’essais → exception TooManyAttemptsException
   * Proposition hors-jeu → exception
4. **Validation UI / UX**
   * Empêcher la sélection de couleurs non autorisées
   * S’assurer que l’utilisateur entre exactement le nombre de pions requis

**b) Architecture et représentation des données**

**i) Lisibilité des données de test**

* **Background** pour créer la partie avec ses paramètres
* **Scenario Outline + Examples** pour tester plusieurs codes et retours de feedback
* **Scénarios ciblés** pour les exceptions (longueur, couleurs invalides, après fin de partie)

**ii) Extensibilité**

* Les StepDefinitions (Given une partie Mastermind, When je propose, Then j’obtiens) restent **modulaires**.
* Le moteur MastermindGame est découplé de l’UI et des tests, prêt à accueillir d’autres variantes (mastermind logique, mastermind multijoueur, etc.).
* Nouveaux jeux basés sur la même architecture (comme Puissance 4) peuvent être ajoutés simplement dans un dossier Features/Mastermind/.

**c) Stratégie BDD et bonnes pratiques**

**i) Langage ubiquitaire**

Termes métier clairs : **code secret**, **proposition**, **pion noir**, **pion blanc**, **palette de couleurs**, **essai**, **partie terminée**.

**ii) Réutilisabilité des StepDefinitions**

* **Given** : une partie Mastermind est configurée, le code secret est “…”
* **When** : le joueur propose “…”
* **Then** : j’obtiens X pions noirs et Y pions blancs, une exception E est levée, la partie est (non) terminée

**iii) Maintenance**

* Dossiers clairs : Features/Initialisation, Features/Propositions, Features/Exceptions
* Scénarios et Steps nommés explicitement
* Code de test et code de production indépendants, facilitant refactoring et ajout de nouvelles règles de jeu.

**TicTacToe**

**a) Analyse et justification des scénarios**

**i) Identification des cas de test**

**Cas nominaux**

* Placer un symbole (X ou O) sur une case vide
* Alterner les tours entre X et O
* Déclarer le gagnant quand une ligne, une colonne ou une diagonale est complétée
* Déclarer un match nul lorsque toutes les cases sont remplies sans gagnant

**Cas limites**

* Tenter de jouer sur la première ou la dernière case
* Gagner au dernier coup
* Déclencher un match nul au dernier coup
* Vérifier la détection du vainqueur dans tous les cas : ligne, colonne, diagonale

**Cas d’erreur / protection**

* Jouer sur une case déjà occupée → lever InvalidMoveException
* Jouer hors de la grille (ex: case (-1,0) ou (3,3)) → lever ArgumentOutOfRangeException
* Jouer alors que la partie est déjà terminée → lever InvalidOperationException

**ii) Priorisation des scénarios**

1. **Fondamentaux**
   * Initialiser une grille vide
   * Définir quel joueur commence (X ou O)
2. **Interaction de base**
   * Jouer un coup valide
   * Alterner automatiquement les joueurs
3. **Fin de partie**
   * Vérifier la victoire de X ou O
   * Match nul
   * Tentative de jouer après la fin → exception
4. **Fonctionnalités auxiliaires**
   * Réinitialiser la grille
   * Afficher l’état de la grille

**b) Architecture et représentation des données**

**i) Lisibilité des données de test**

* Background pour définir une grille vide ou pré-remplie
* Scenario Outline pour tester les cas de victoire (lignes, colonnes, diagonales)
* Scénarios séparés pour erreurs (InvalidMoveException, ArgumentOutOfRangeException)

**ii) Extensibilité**

* StepDefinitions génériques : Given une grille vide, When je joue en (x,y), Then le joueur courant est, Then la partie est gagnée
* Moteur de jeu (TicTacToeGame) séparé, logique isolée
* Possibilité d’étendre vers un mode IA ou multijoueur réseau sans modifier les scénarios existants

**c) Stratégie BDD et bonnes pratiques**

**i) Langage ubiquitaire**

Termes utilisés : grille, case, joueur, X, O, match nul, victoire, coup, partie terminée

**ii) Réutilisabilité des StepDefinitions**

* Given : grille vide ou état initial, coups précédents
* When : joueur joue en (x,y)
* Then : grille attendue, message de victoire ou nul, exception levée

**iii) Maintenance**

* Dossiers de .feature : Initialisation, Partie, Victoire, Erreurs
* Organisation claire des Steps par fonctionnalité (partie, joueur, erreurs)
* Nommage explicite des scénarios : Victoire par ligne, Match nul, Erreur : jouer sur une case occupée

**Snake**

**a) Analyse et justification des scénarios**

**i) Identification des cas de test**

**Cas nominaux**

* Démarrer une nouvelle partie
* Faire avancer le serpent dans une direction
* Manger une pomme → augmenter la taille et le score

**Cas limites**

* Le serpent mange une pomme juste avant un mur
* Le serpent est à une case du bord ou de lui-même
* Gagner un point exactement après une collision

**Cas d’erreur / protection**

* Collision avec un mur → fin de partie
* Collision avec lui-même → fin de partie
* Tentative de changer de direction vers l’arrière (ex: de droite à gauche) → ignorée ou rejetée

**ii) Priorisation des scénarios**

1. **Fondamentaux**
   * Initialisation de la partie (taille de la grille, position de départ)
   * Mouvement simple du serpent
2. **Interaction de base**
   * Le serpent mange une pomme
   * Augmentation du score et de la taille
   * Apparition d’une nouvelle pomme
3. **Fin de partie**
   * Collision avec un mur
   * Collision avec soi-même
   * Direction inverse interdite
4. **Fonctionnalités auxiliaires**
   * Réinitialisation de la partie
   * Affichage du score final

**b) Architecture et représentation des données**

**i) Lisibilité des données de test**

* Background pour créer la grille, position du serpent, direction
* Scenario Outline pour tester plusieurs positions / pommes / directions
* Scénarios unitaires pour les cas de collision ou règles interdites

**ii) Extensibilité**

* StepDefinitions réutilisables : Given une partie de Snake, When le joueur dirige le serpent vers ..., Then la partie est terminée
* Moteur (SnakeManager) indépendant de l’UI
* Possibilité d’ajouter des niveaux, obstacles, ou bonus via de nouveaux scénarios

**c) Stratégie BDD et bonnes pratiques**

**i) Langage ubiquitaire**

Termes métier clairs : serpent, direction, pomme, mur, collision, score, tête, corps

**ii) Réutilisabilité des StepDefinitions**

* Given : position initiale du serpent, pomme placée, score initial
* When : direction prise, mouvement
* Then : état du jeu (score, longueur, fin de partie), collision, exception éventuelle

**iii) Maintenance**

* Organisation en .feature : Déplacement, Pomme, Collision, Réinitialisation
* Fichiers Steps bien séparés : DéplacementSteps, CollisionSteps, etc.
* Chaque scénario isolé permet d’identifier rapidement l’objectif métier testé