Ecole Publique d'Ingénieurs en 3 ans

Rapport

PROJET GENIE LOGICIEL

Le 16 octobre 2023

Guillain Le-Goff	guillain.le-goff@ecole.ensicaen.fr
Leo Bretagne	leo.bretagne@ecole.ensicaen.fr
Lucas Ollier	lucas.ollier@ecole.ensicaen.fr
Clement Cartro	clement.cartro@ecole.ensicaen.fr
Moris-Lorys Kamgang	moris-lorys.kamgang@ecole.ensicaen.fr
Achraf Aboulakjam	achraf.aboulakjam@ecole.ensicaen.fr
Hamid Ben-Omar	hamid.ben-omar@ecole.ensicaen.fr

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

www.ensicaen.fr

TABLE DES MATIERES

TABLE DES FIGURES

INTRODUCTION

Ce projet vise à mettre en pratique les principes du génie logiciel et des patrons de conception pour développer un jeu de plateau où chaque joueur incarne un étudiant en formation à l'école. Le jeu s'inspire du "jeu de l'oie" mais introduit des éléments spécifiques liés à la formation, la vie étudiante et à la personnalité des étudiants. L'objectif est de créer un logiciel extensible et réutilisable, en accordant une attention particulière à la qualité de la conception grâce aux principes SOLID et aux patrons de conception. Les exigences du client qui ont été implémentées incluent une interface graphique multilingue, la création d'un plateau de jeu avec diverses cases à effets, la gestion des caractéristiques des étudiants et la génération d'un classement en fonction de la position à la fin de la partie. Le prolongement de ce projet permettrait d'ajouter la lecture d'un fichier JSON pour configurer le plateau, de réaliser une requête vers une ressource en ligne pour déterminer le salaire moyen d'un ingénieur, ainsi que d'enregistrer les résultats dans une base de données.

ORGANISATION DU PROJET

1. Diagramme des cas d'utilisation

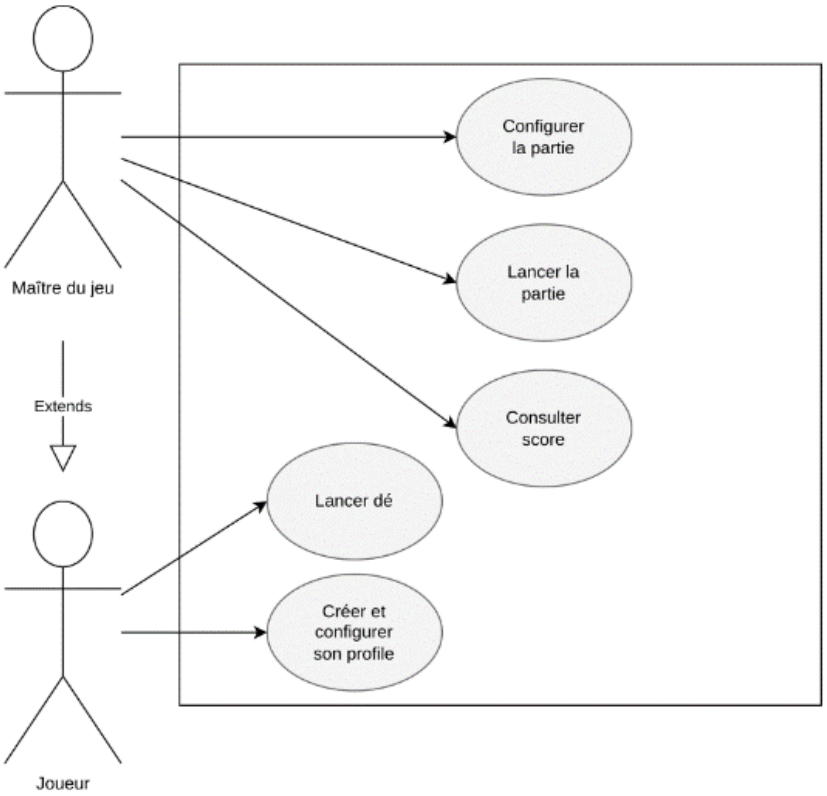

Figure 1 : Diagramme des cas d'utilisation

2. Analyse des risques

Risques du projet	Gravité	Probabilité	Criticité
Manque de connaissances techniques : Java, JavaFX, Git, JSON, XML,	Majeure	Modérée	Élevée
Différents environnements de développement (Windows, Linux, Mac)	Significatif	Certaine	Élevée
Liste des tâches pas assez exhaustive	Significatif	Modérée	Moyen
Répartition inégales des tâches	Significatif	Probable	Moyen
Difficultés à estimer la durée des tâches	Majeure	Très probable	Très élevée
Communication insufisante entre les membres de l'équipe	Sévère	Modérée	Très élevée
Indisponibilité d'un membre	Significatif	Improbable	Moyen
Fonctionnalités trop complexes à mettre en œuvre	Majeure	Modérée	Élevée
Fonctionnalités défectueuses	Sévère	Probable	Très élevée
Produire du code incompatible avec le code des autres membres	Sévère	Probable	Très élevée

Figure 2 : Analyse des risques

Risques du projet	Solutions
Manque de connaissances techniques : Java, JavaFX, Git, JSON, XM	Se former à ces technologies en dehors des séances, demander de l'aide aux membres
Différents environnements de développement (Windows, Linux, Mac)	Discuter de son environnement avec le responsable de version pour vérifier la compatibilité
Liste des tâches pas assez exhaustive	Prendre le temps de bien définir les objectifs du projet et son fonctionnement
Répartition inégales des tâches	Réviser et ajuster régulièrement la répartition des tâches
Difficultés à estimer la durée des tâches	S'appuyer sur ses expériences en développement, diviser les tâches en sous tâches
Communication insufisante entre les membres de l'équipe	Utilisation d'un groupe de messagerie instantannée pour communiquer, organiser des bilans réguliers et des pauses pour discuter
Indisponibilité d'un membre	Déléguer le travail à réaliser aux membres ayant le moins de travail à priori, Prévenir le reste de l'équipe si possible
Fonctionnalités trop complexes à mettre en œuvre	Bien évaluer la difficulté de la fonctionnalité en fonction du temps disponible, Simplifier la fonctionnalité, la découper en plusieurs sous tâches faciles
Fonctionnalités défectueuses	Mettre en place des tests rigoureux pour détecter les problèmes, Effectuer du peer reviewing pour identifier les erreurs potentielles.
Produire du code incompatible avec le code des autres membres	Établir des normes de codage claires et demander des conseils/consignes au responsable de version

Figure 3 : Solutions des risques

3. Diagramme de package

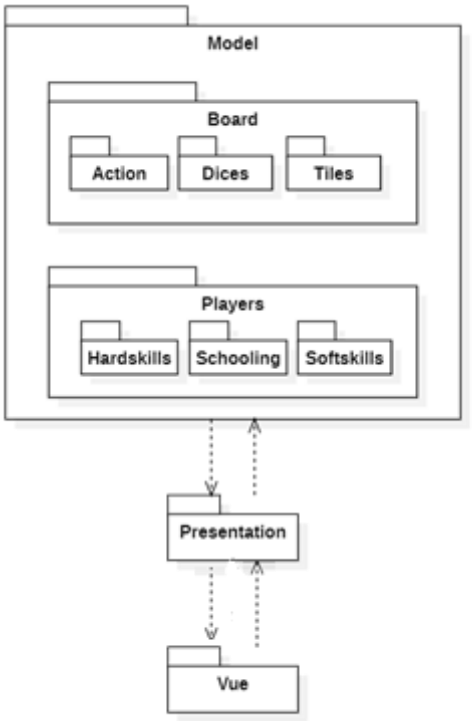

Figure 4 : Diagramme de package

OBJECTIFS

1. MVP 1

- Création d'un plateau minimaliste de 5 cases sans effets et d'un bouton pour lancer le dé

- Création d'un joueur avec un SoftSkill attribué aléatoirement

- Création d'un menu de création de joueur

- Pas de HardSkills pour l'instant

- Déplacement du joueur d'autant de cases que le résultat du dé

- Affichage du résultat du dé sous forme d'image sur le plateau de jeu

2. MVP 2

- Avoir un plateau avec des cases à effet sur les compétences, et sur le déplacement du joueur

- Disponibilité du jeu en deux langues (Français – Anglais)

- Influence du SoftSkill sur le résultat du dé

- Ajout de HardSkills

- Affichage du tableau de résultat à la fin du jeu

- Amélioration de la qualité logicielle de certaines classes : diviser des classes en sous classes à responsabilité unique, utiliser des patrons de conception pour rendre le logiciel plus ouvert aux extensions

CONCEPTION UML

1. Structure générale

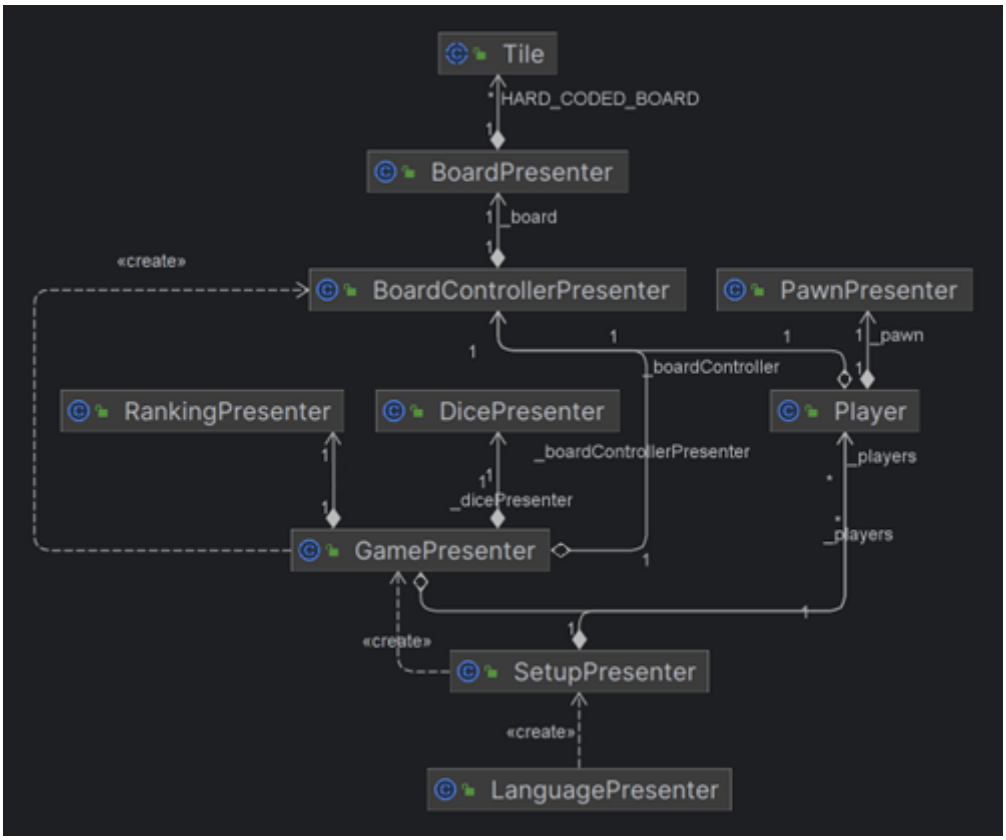

Figure 5 : Structure principale de la boucle de jeu

1.1. Patron d'architecture

Le patron d'architecture est le MVP, dont les principes ont été essentiellement respectés sauf pour les exemples suivants :

- La classe Player a bien trop de responsabilités et aurait au moins dû être séparée en un Presenter et un Model.
- Les interactions entre Action, Tile et Player conduisent à ce que les instances d'Action communiquent avec le Player dans ses caractéristiques qui devraient être dans un présentateur associé.

Il est également à mentionner qu'au cours de ce projet nous n'avons pas suivi les principes de conception en paquets.

1.2. Model

1.2.1. Board

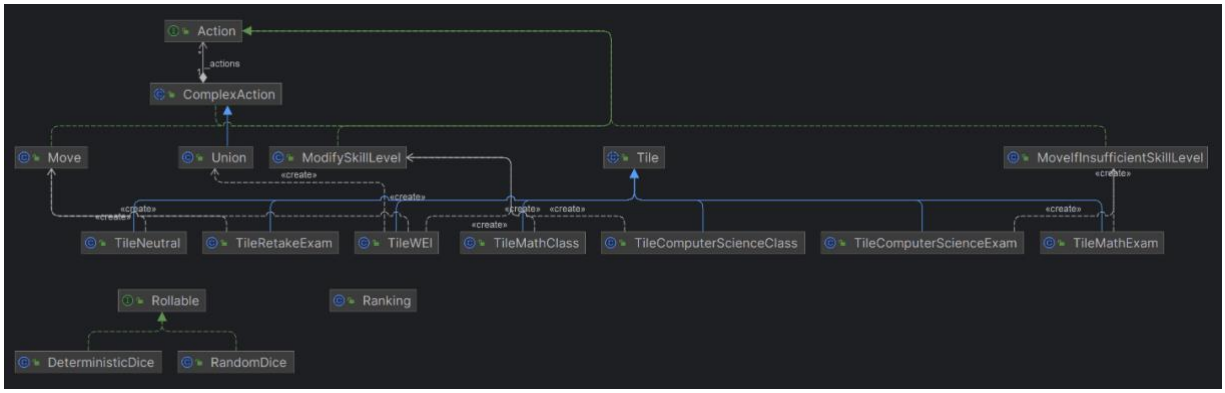

Figure 6 : Package Board

Le code a été conçu de façon à être le plus SOLID possible, il a été notamment important de garantir une bonne développabilité en utilisant des interfaces ou des classes abstraites pour encapsuler ce qui varie et programmer pour des interfaces plutôt que pour leurs implémentations. C'est notamment le cas dans le package Board avec la classe abstraite Tile et l'interface Action. Pour être le plus général possible dans les combinaisons d'actions, un patron d'architecture composite a même été utilisé.

1.2.2. Player

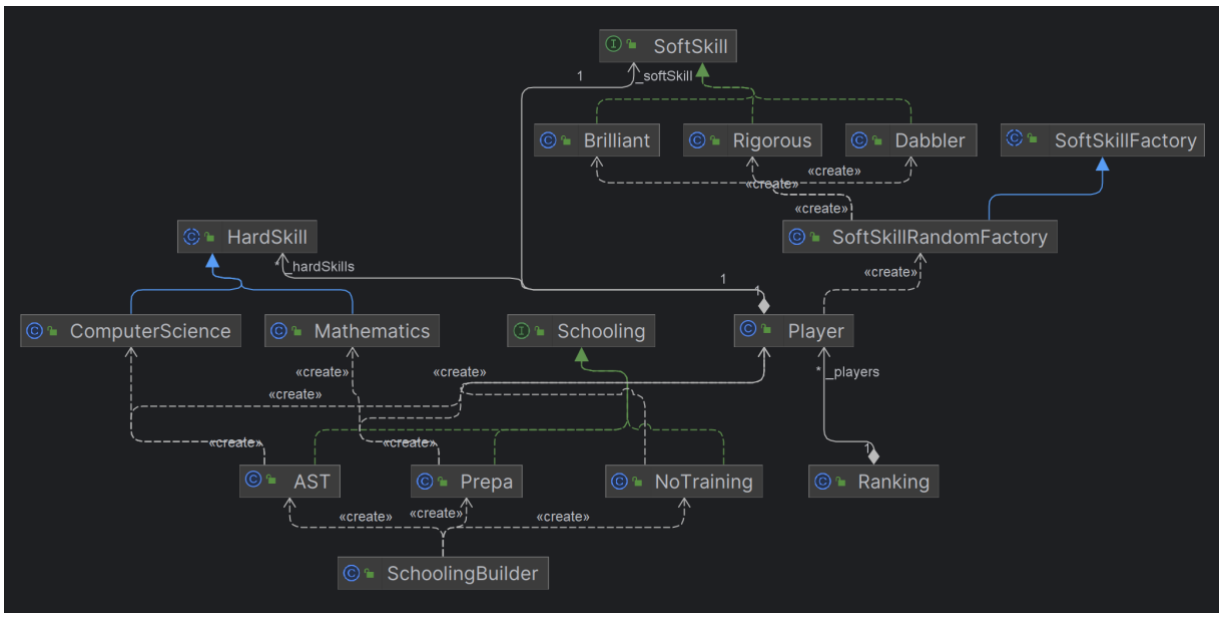

Figure 7 : Package Player

Dans ces packages, nombre de Fabriques ont été employées pour déléguer les responsabilités de création des joueurs selon leur formation (Schooling) et des SoftSkills toujours pour favoriser la développabilité.

Néanmoins nous avons repéré quelques défauts. En effet, les Schooling connaissent les HardSkills du Player, ce qui rend cette partie du code "ouvertes aux modifications" et les dépendances sont dans le mauvais sens. Il aurait été préférable de passer par un patron monteur ce qui aurait corrigé les deux problèmes.

Il est aussi notable qu'au cours du projet, et probablement à cause d'un manque de communication interne, des classes suivant l'anti-patron Objet Omniscient sont apparues. Afin de ne pas nuire au principe de responsabilité unique, il a été nécessaire d'arrêter momentanément le développement pour éclater les responsabilités dans des classes appropriées. Ceci est vraiment problématique car les tests unitaires associés doivent être modifiés et la procédure freine le développement de nouvelles fonctionnalités.

2. Patron de conception

2.1. Builder

En remarquant que la classe RandomDice possédait 4 constructeurs légèrement différents pour paramétrer ses trois attributs, nous avons décidé de mettre en place un Monteur.

Le monteur a été implémenté de la façon suivante :

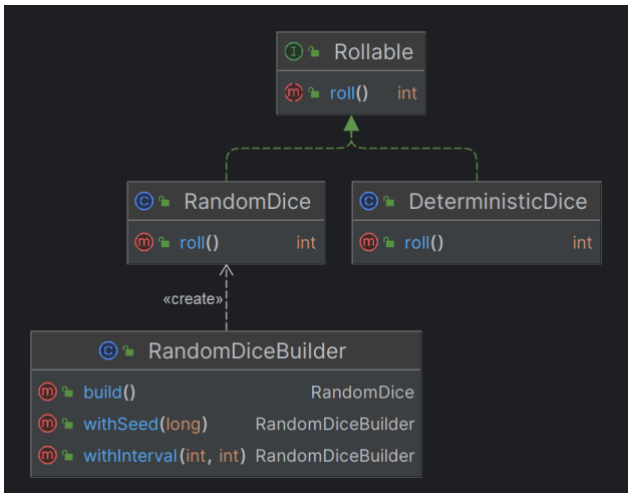

Figure 8 : Patron monteur

2.2. Fabrique

Puisque la classe Player commençait à avoir beaucoup de responsabilités, nous avons choisi de déléguer la responsabilité de la création des joueurs à une fabrique simple. De même pour la classe Schooling.

Celles-ci ont été implémentées de la manière suivante :

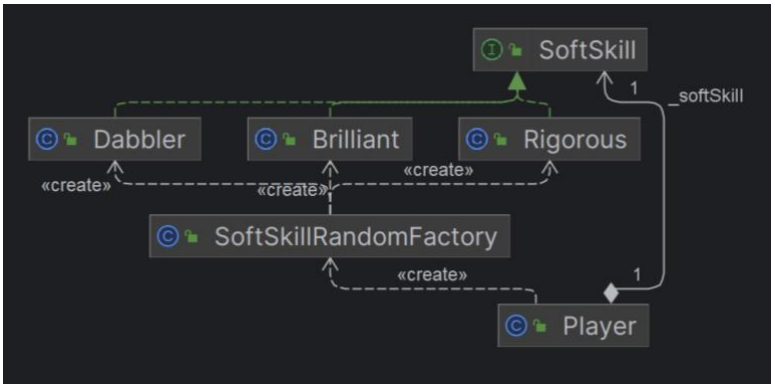

Figure 9 : Patron fabrique

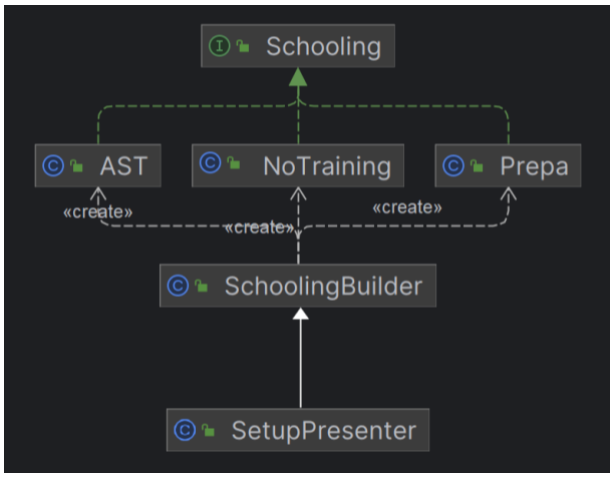

Figure 10 : Patron fabrique

2.3. Composite

Puisque nous avons réparti les effets des cases sous forme d'actions à appliquer au joueur, il s'est avéré utile de créer des actions complexes pour combiner des actions simples.

Le composite est implémenté de cette façon :

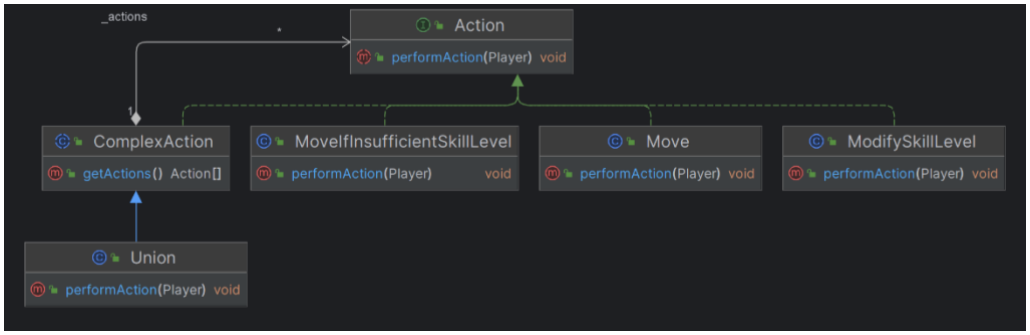

Figure 11 : Patron composite

2.4.Messager

Puisqu'il fallait régulièrement échanger des couples d'entiers pour l'affichage des coordonnées de notre pion, nous utilisons un messager constitué des coordonnées du pion sur la zone de dessin.

Le messager a été implémenté de cette façon :

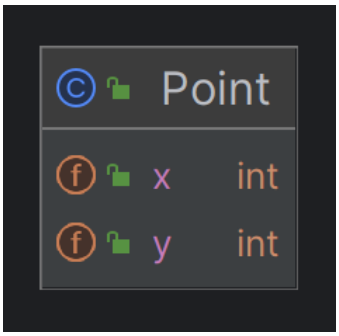

Figure 12 : Patron messager

COUVERTURE DE TEST

1. Objectifs de la couverture de test

Afin d'assurer la qualité du logiciel, il était nécessaire de mettre en œuvre une série de tests pour éliminer les éventuels bugs susceptibles de nuire à l'expérience utilisateur et au développement de nouvelles fonctionnalités. Ces tests permettront de s'assurer que le logiciel répond aux spécifications et aux exigences fonctionnelles et techniques définies.

Le projet ayant une architecture MVP (Model – View - Presenter), les deux parties que nous avons testé sont les parties contenant de la logique : Model et Présenter. Pour mener à bien nos tests nous avons utilisé JUnit5 et Mockito.

2. Méthodologie des tests

La méthodologie de test que nous avons utilisée consiste en plusieurs étapes :

1. Identification des fonctionnalités à tester.
2. Définition des limites du test : il ne faut tester que ce que fait la méthode, et non pas les sous-méthodes qu'elle appelle.
3. Conception des scénarios de test.
4. Exécution des tests et analyse des résultats.
5. Identification et correction des problèmes.
6. Répétition et validation des tests.

3. Commentaires

3.1. Responsabilité d'un test

Lorsque nous avons commencé à implémenter les premiers tests, nous avons réalisé que certains de nos tests dépendaient d'autres méthodes que celle testée, nous avons donc par la suite fait attention à tester uniquement ce que fait concrètement cette méthode à l'aide du framework mockito, qui nous permet en autres de vérifier que certaines « sous » méthodes sont appelées avec les bons arguments, ainsi que de configurer des comportements pour un objet simulé « mock ».

3.2. Exemple de détection d'erreur

Au cours d'une revue de code de test, nous avons remarqué que le test ayant pour but de vérifier que le dé renvoie des valeurs dans le bon intervalle ne vérifiait qu'un seul lancer de la méthode de lancer de dé. Nous avons donc augmenté le nombre de lancers, ce qui a mis en évidence un bug de la méthode de lancer de dé. Nous avons ensuite corrigé ce bug.

ANALYSE DES RESULTATS

Dans sa deuxième version, notre projet de jeu de l'oie possède d'ores-et-déjà des caractéristiques intéressantes en termes de visuel et de jouabilité. Toutefois seul le client peut évaluer la véritable valeur du produit qui a été conçu pour lui. Nous nous contenterons donc de détailler les caractéristiques techniques du prototype que nous avons obtenu.

1. Méthode

Au cours du déroulement du projet, nous avons repéré 5 problèmes importants qu'il serait impératif de corriger avant d'en débuter un nouveau :

- Trop de modélisation à l'avance.

- Peu de contrôle sur les productions individuelles des membres de l'équipe.

- Peu de partage de connaissances.

- Une mauvaise gestion des priorités.

- Problèmes de communication.

1.1. Trop de modélisation à l'avance

Problèmes :

- Une partie de ce qui a été pensé ne sera pas utilisé.

Certaines fonctionnalités anticipées et non implémentés conduisent à du code non utilisé (non YAGNI).

- Si la modélisation originale posait un problème, c'est la structure entière qui en subit les conséquences, et notamment les tests unitaires qui sont à récrire.

- Certains concepts semblaient SOLID mais se sont révélés non testables à l'implémentation.

Solution envisageable :

- Penser les modélisations pour l'itération en cours lors de la définition des tâches. Tout en ayant une vague connaissance des fonctionnalités à venir grâce au diagramme métier initial, cela permet de savoir quelles sont les fonctionnalités susceptibles d'être étendues.

1.2. Peu de contrôle sur les productions individuelles des membres de l'équipe.

Problèmes :

- Certains oublient de réaliser des tests unitaires, ou de terminer leur refonte de code.

- Le choix des noms de variables et de méthodes peut mener à des incompréhensions et sont parfois difficilement réversibles.

Solution envisageable :

- Pratiquer la revue par les pairs. Une pratique courante en entreprise est de faire approuver les modifications d'une branche par les autres membres du groupe avant que celle-ci ne soit fusionnée.

1.3. Peu de partage de connaissances

Problèmes :

- Les membres du groupe se sont rapidement retrouvé spécialisés (en FXML, en MVP, en git, en tests unitaires…) et par conséquent se sont contentés d'effectuer des tâches qu'ils connaissaient déjà.

- Lorsqu'aucun membre d'une sous-équipe ne possède les compétences requises pour la tâche aucun progrès significatif ne peut être effectué.

Solutions envisageables :

- Diversifier davantage les sous-équipes pour partager les connaissances.

- Changer de tâche ou modifier les équipes si une tâche se révèle au-delà des compétences de ses membres.

1.4. Mauvaise gestion des priorités

Problèmes :

- Vouloir trop en faire pendant la même itération ce qui a conduit à négliger la refonte de code.

- Les remarques du client non nécessairement prises en compte au profit de fonctionnalités estimées plus importante.

Solutions envisageables :

- Revoir les espérances à la baisse pour le MVP (minimum viable product).

- Constater les erreurs de parcours à la fin de l'itération avec le client et les prendre en compte pour l'itération suivante, le client étant le seul capable d'évaluer la qualité du logiciel.

1.5. Problèmes de communication

Problèmes :

- Certains se sont retrouvés exclus de certaines discussions pour terminer une tâche, ce qui peut provoquer un désintérêt pour le projet.

- Structure du projet connu que par un nombre limité de membres notamment au début du projet.

- Parfois il suffit de la bonne personne au bon moment pour débloquer une équipe mais c'est difficile de le savoir.

Solutions envisageables :

- Faire un tour des autres équipes une fois une tâche terminée pourrait apporter une aide qui serait bénéfique au projet.

- Expliquer plus en profondeur l'objectif de l'itération pour que chacun comprenne mieux son implication dans les tâches à venir.

2. Aspect technique

Malheureusement la configuration ne nous permettra pas de nous interroger sur la maintenabilité et la réutilisation de notre production, toutefois quelques points positifs sont à remarquer concernant la développabilité.

2.1. Aspect développabilité

Toutefois, puisque le projet vise une conception logicielle SOLID, nous pouvons constater que celui-ci possède en effet des facilités de développabilité, ce qui a été remarqué lors de l'ajout de nouveaux types de cases et d'effets lors de la deuxième itération.

Les tests unitaires permettent également de faciliter la détection d'erreurs lors de l'implémentation de nouvelles fonctionnalités et ont été pratiques à maintes reprises afin de ne pas nuire aux fonctionnalités déjà existantes.

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Ecole Publique d'Ingénieurs en 3 ans

6 boulevard Maréchal Juin, CS 45053
14050 CAEN cedex 04