Année universitaire 2024-2025

Rapport Projet CAPI

Université Lumière Lyon 2

Pierrick Dennemont
Loïc Babolat

Introduction

L'objectif de ce projet était d'implémenter le jeu du « Planning Poker » tout en respectant les principes de l'intégration continue appris lors des cours de Conception agiles. Cela intègre la mise en place de tests, permettant d'éviter la régression dans le code mais aussi la génération automatique de la documentation. Ces deux aspects seront approfondis par la suite.

Notre application, accessible uniquement par un ordinateur local, permet à l'utilisateur de renseigner le nombre de joueurs et de fonctionnalités pour la partie. Il est aussi en mesure de nommer sa partie, pour plus de clarté. On a pris la décision d'intégrer pour notre projet les modes de jeu suivants :

- Unanimité
- Moyenne
- Médiane

Choix techniques

Préférences de développement

Langage de programmation

Pour notre projet on a décidé d'utiliser le langage natif JavaScript. Celui-ci correspondait le plus à nos besoins et à nos exigences. En effet, il nous semble plus optimal de travailler avec un langage orienté web pour développer des applications. D'une autre part, JavaScript facilite grandement la mise en place d'une interface dynamique et interactive, celle-ci étant essentielle et omniprésente dans les applications développés ces dernières années. De plus, sa compatibilité avec Node.js était un facteur important dans notre choix, car tous deux ont été habitués au cours de nos différents travaux à manipuler cette technologie. Le choix du langage JavaScript est donc avant tout un choix résultant de nos expériences antérieures, tout en prenant en compte la fait que celui facilite une interface dynamique pour l'utilisateur.

Architecture

L'architecture de notre projet repose beaucoup sur le modèle MVC (Modèle, Vue, Contrôleur). Celui-ci divise une application en trois couches distinctes : le Modèle, qui gère les données, la logique métier et les interactions avec la base de données ; la Vue, qui est responsable de l'affichage et de l'interface utilisateur et le Contrôleur, qui agit comme un intermédiaire, gérant les interactions utilisateur, les requêtes, et orchestrant la communication entre le Modèle et la Vue. Néanmoins il est important de préciser que l'architecture de notre projet n'est pas totalement similaire au MVC. En effet, le manque de temps nous a contraint à délaisser le développement de la couche Contrôleur. Ainsi, la récupération ou la création des données, via les classes, n'est pas isolé du reste du code. Cet aspect est un élément à considérer pour une potentielle amélioration de notre code, permettant ainsi de mieux séparer les différentes fonctionnalités du code.

On peut toutefois, malgré l'absence de la couche Contrôleur, proposer une architecture structurée, en conservant la couche Vue et Modèle. En effet, chacune des couches est séparée en un dossier distinct, permettant davantage de clarté dans la structure de notre projet. Le dossier « data » regroupe l'ensemble des classes et de leurs fonctions associées. Cela

comprend aussi l'utilisation de méthodes propres au « Local Storage » : une fonctionnalité du navigateur web qui permet de stocker des données localement, directement sur l'appareil de l'utilisateur, sous forme de paires clé-valeur. Le « Local Storage », jouant un rôle majeur dans notre projet, est ainsi structuré dans un fichier, pour plus de clarté.

La couche « Vue », est quant à elle, stocké dans le dossier « views ». Chaque page web de notre application est comprise dans un sous-dossier, nommé en fonction de sa fonctionnalité. Celui-ci comprend le fichier html mais aussi le fichier JavaScript associée, et pour certaines pages un fichier CSS associé. L'index et le fichier CSS sont laissés à la racine du dossier, pour plus de cohérence. Enfin, les tests, étant réalisés sur les classes et leurs fonctionnalités en elles-mêmes, sont disponible dans la couche « Modèle », soit le dans dossier « data ».

LocalStorage

Pour garder les données en mémoire entre les différentes pages, nous utilisons l'API localStorage qui est native en JavaScript. Cette API nous permet de stocker les données sur le navigateur, pour ne pas avoir à passer toutes les données dans l'URL lorsque nous passons d'une page à l'autre. Nous avons seulement besoin de passer l'identifiant correspondant à la donnée que nous voulons récupérer.

Explication du code

Dans cette partie, on se propose d'expliquer plus en détail les aspects qui nous semblent essentiels ou important de détailler dans notre code. Chaque explication est susceptible d'être accompagné d'une capture d'écran du code, pour davantage de lisibilité.

Création des classes

Cette partie proposera des détails sur la conception des classes permettant de structurer nos données. Notre projet comporte trois classes différentes, reliés entre elles :

- La classe User : celle-ci structure les données des différents joueurs. Chaque joueur dispose d'un id et d'un nom. L'id permet notamment de reconnaitre un joueur avec un nom identique à un autre participant.
- La classe Backlog : elle permet de stocker les données concernant les fonctionnalités à noter. Chaque fonctionnalité possède un identifiant, un titre, une description, une valeur finale (finalRate) ainsi qu'un tableau de notes (rate). L'attribut rate est un tableau d'objet RateObject, celui-ci comprenant une valeur et un nom d'utilisateur. Cet objet n'existe pas en réalité dans notre code, mais est détaillé dans la documentation pour mieux comprendre le fonctionnement de la classe Backlog. Enfin, cet objet dispose d'un champ isFirstTurn, permettant de gérer le mode de jeu du premier tour.
- La classe Game : celle-ci permettra de générer une partie et de stocker ses données. Chaque partie admet un id, un nom, un mode difficulté, une liste de joueurs (tableau d'User) et une liste de fonctionnalité (liste de Backlog). Les fonctions permettant l'avancé de la partie se feront directement grâce à cette classe.

Par soucis de précaution, chaque id est généré automatiquement avec le module UUID, permettant de générer des identifiants uniques sous forme de chaînes de caractères. Le diagramme ci-dessous reprend la description des classes décrites ci-dessus :

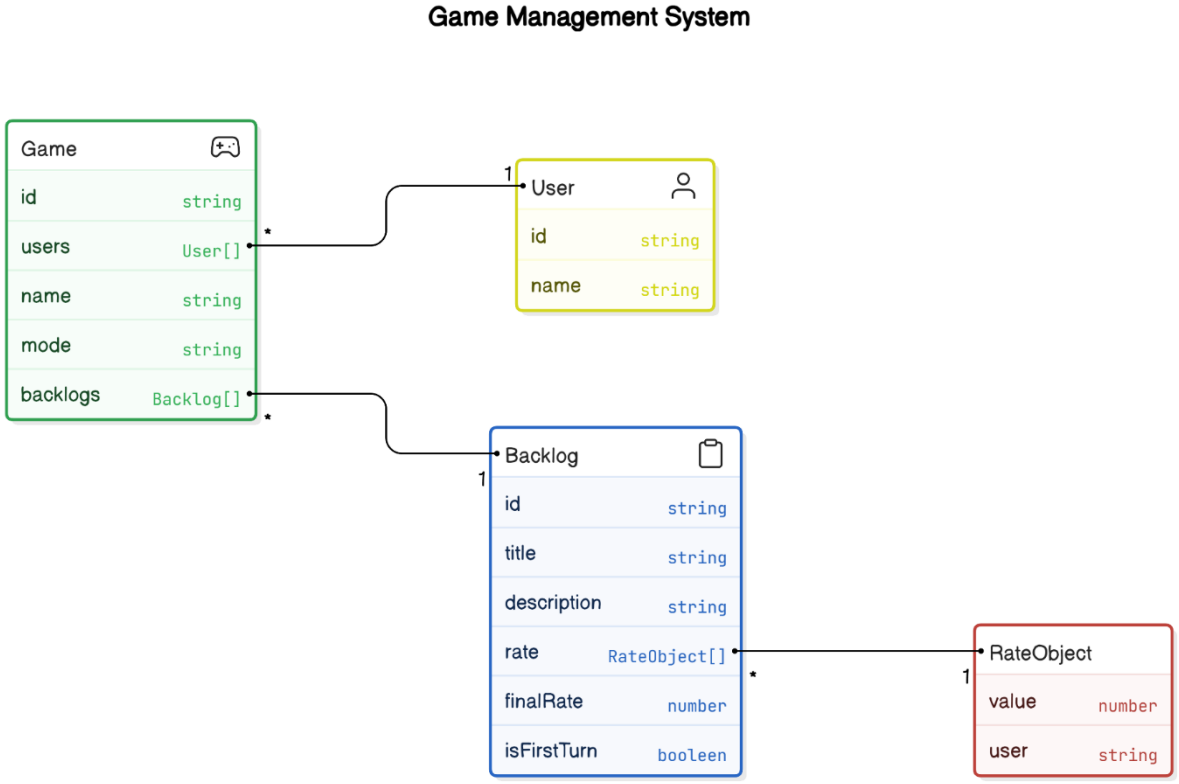

Diagramme de classe

Toute ces classes disposent de Getter et de Setter, pour chacun de leurs attributs. Les utilisent le système de Local Storage pour accéder ou stocker des données. Pour chaque Setter, l'attribut est enregistré dans le LocalStorage, avec une clé unique et en lien avec le nom de l'attribut. On peut par exemple prendre le cas du titre d'un Backlog, dont le code est affiché ci-dessous :

```
/**
 * Instancie le titre de la fonctionnalité
 * @param {string} title Titre de la fonctionnalité
 */
set title(title) {
    this.#title = title
    setItem(`backlogTitle${this.#id}`, this.#title)
}
```

Exemple de Getter

On peut voir dans cet exemple que l'élément est enregistré dans le LocalStorage avec la clé « backlogTitle … ». Cette méthode est plus explicite qu'uniquement appeler la fonction avec un simple identifiant.

Certains Getters de la classe Game méritent d'être plus approfondi. En effet, ceux-ci sont différents de ceux présents dans les autres classes, en particulier pour les attributs « backlogs » et « users ». Cela s'explique par le fait que ces deux champs sont des tableaux regroupent des types particuliers. Ainsi, la valeur ou stockée dans le Local Storage diffère des autres attributs.

Pour une unique clé, le Local Storage stockerait dans la valeur un tableau d'identifiant (id), comme le montre la capture d'écran ci-dessous. Cela conserve les conditions d'intégrité établit plus tôt.

```
/**
 * Retourne la liste des backlogs
 * @returns {Backlog[]} L'ensemble des backlogs
 */
get backlogs() {
    return this.#backlogs
}
/**
 * Instancie les backlogs de la game
 * @param {Backlog[]} backlogs Liste des backlogs
 */
set backlogs(backlogs) {
    if (backlogs != null) {
        this.#backlogs = backlogs
        let ids = []
        for (let backlog of this.#backlogs) {
            ids.push(backlog.id)
        }
        setItem(`gameBacklogs${this.#id}`, ids)
    }
}
```

Classe Game

Cette partie proposera de détailler les différentes fonctions de la classe « Game », celle-ci permettant de contrôler le déroulement du jeu.

Gestion des modes de jeu

Il convient de rappeler que le premier tour de vote, peu importe le mode de jeu sélectionné, doit se faire en mode « unanimité ». Ainsi, la classe « Game » doit pouvoir gérer le mode de jeu courant, qui peut avoir une valeur différente de son attribut « mode ». Pour cela, on a développé la fonction « getCurrentMode », qui applique le mode de jeu « unanimité » au backlog en cours les utilisateurs en sont à la phase du premier tour. Cela est géré avec la valeur du booléen « isFirstRate » de l'objet « Backlog ».

```
/**
 * Retourne le mode de vote actuel de la game, retourne donc unanimite si c'est
 * le premier tour de vote, meme si le mode est différent
 * @param {Backlog} backlog
 * @returns {{'moyenne'|'mediane'|'unanimite'}}
 */
getCurrentMode(backlog) {
    if (backlog.isFirstTurn) {
        return "unanimite"
    } else {
        return this.#mode
    }
}
```

Fonction de la classe Game 1

Ainsi, le mode de jeu est géré en fonction du numéro du tour (s'il est différent de 1 ou non) et du mode de jeu choisi pour la partie. On peut dès à présent expliquer en détail la gestion des notes pour chaque fonctionnalité.

Gestion des notes

On proposera l'explication pour uniquement un des modes de jeu développé. En effet, la méthode appliquée reste similaire pour chaque mode de jeu implémenté, à l'exception de la récupération de la note finale. Dans notre rapport, on explicitera la fonction « unanimityRate », permettant, comme son nom l'indique, la gestion de la note finale en fonction du mode de jeu « unanimité ».

Cette fonction prend en paramètre un objet de type « Backlog ». Elle nécessite que l'ensemble des joueurs ait soumis un vote. En effet, le stockage des notes individuelles réside dans l'attribut « rates » du backlog passé en paramètre. Ainsi, la fonction, pour chaque valeur du tableau « rates », conservera uniquement la valeur du vote de chaque joueur. Un traitement spécifique à chaque mode de jeu est ensuite implémenté pour calculer la note finale de la fonctionnalité. Cette valeur est ensuite stocké dans l'attribut « finalRate » du backlog sollicité. Pour rappel, les différents backlogs sont associés à l'identifiant de la partie, permettant ainsi la récupération des champs de chaque backlog grâce à un identifiant. Ci-dessous est affiché le code correspondant à la fonction « unanimityRate ». Les fonctions « averageRate » et « medianRate » partage la même logique de programmation.

```
/**
 * Retourne un boolean pour savoir si la note est votée à l'unanimité, et donne
 * la valeur de la note à finalRate
 * @param {Backlog} backlog Le backlog à vérifier
 */
unanimtyRate(backlog) {
    const rates = backlog.rates.map(rate => rate.value)
    if (rates.every(rate => rate === rates[0])) {
        backlog.finalRate = rates[0]
    } else {
        // Si la note n'est pas votée à l'unanimité, on met la note finale à -1 pour dire
        // que le vote est en cours
        backlog.finalRate = -1
    }
}
```

Fonction de la classe Game 2

Il suffit, à partir des fonctions présentées ci-dessus, de les appeler en fonction du tour et du mode de jeu en cour. Si le vote du backlog n'est plus au stade du premier tout, alors on appelle la fonction permettant de gérer la note finale du backlog, en fonction du mode de jeu renseigné pour la partie (attribut « mode » de la classe Game). Sinon, le mode de jeu par défaut est le mode « unanimité ». Par défaut, la note finale du backlog est initialisé à -2, ce qui équivaut à dire que la fonction n'a pas encore été voté. La valeur prend la valeur -2 lorsque les utilisateurs ont effectué un tour de jeu et enfin la valeur calculé par les fonctions « unanimityRate », etc lorsque la fonctionnalité a fini d'être votée et discutée. La fonction « isFinalRate », présenté ci-dessous, gère la notation en fonction du mode de jeu en cours, ainsi que la gestion des tours. A chaque fin de tour, l'utilisateur est renvoyé dans la page « game_resume.html », celle-ci affichant la liste

des backlogs et leur statut.

```
/**
 * Retourne true si le backlog est voté, sinon false
 * @param {Backlog} backlog
 * @returns {boolean}
 */
isRateOver(backlog) {
    if (!backlog.isFirstTurn) {
        switch (this.#mode) {
            case "moyenne":
                this.averageRate(backlog)
                break
            case "mediane":
                this.medianRate(backlog)
                break
            case "unanimite":
                this.unanimtyRate(backlog)
                break
            default:
                throw new Error("Le mode de vote n'est pas reconnnaissable")
        }
    } else {
        this.unanimtyRate(backlog)
        backlog.passedFirstTurn()
    }
    if (this.#backlogs.every(backlog => backlog.finalRate !== -1)) {
        return true
    } else {
        return false
    }
}
```

Fonction de la classe Game 3

Initialisation d'une partie

L'utilisateur a deux possibilité pour créer une nouvelle partie. La première est d'utiliser l'interface présent dans notre application, il est ainsi guidé étape par étape pour la création d'un objet « Game ». La seconde option est d'importer un fichier json et de créer la partie à partir de celui-ci. Le fichier doit cependant être bien structuré pour permettre la création de la partie. On détailler a dans cette partie le fonctionnement de ces deux méthodes.

Par saisie de l'utilisateur

La fonction statique « initFromId » de la classe « Game initialise une instance de la classe à partir d'un identifiant unique (« id »). Elle récupère les données liées à cet identifiant dans le LocalStorage, crée les instances nécessaires de « User » et « Backlog, puis retourne un nouvel objet de type « Game » avec ces données. Les données stockées dans le LocalStorage ont été créés au fur et à mesure des interactions de l'utilisateur avec les différentes pages de notre application. Le paramètre « id » de la fonction est un identifiant unique pour la game. Cet identifiant est utilisé pour récupérer les données associées à la game dans le local storage.

```
/**
 * Pour initialiser une game à partir de son id, en récupérant les données liées à cet id dans
 * le local storage
 * @param {string} id Un identifiant unique pour la game
 * @returns {Game}
 */
static initFromId(id) {
    let usersId = getItem(`gameUsers${id}`)
    let users = undefined
    if (usersId != null) {
        users = []
        for (let userId of usersId) {
            users.push(User.initFromId(userId))
        }
    }
    let backlogsId = getItem(`gameBacklogs${id}`)
    let backlogs = undefined
    if (backlogsId != null) {
        backlogs = []
        for (let backlogId of backlogsId) {
            backlogs.push(Backlog.initFromId(backlogId))
        }
    }
    return new Game(getItem(`gameName${id}`), getItem(`gameMode${id}`), users, backlogs, id)
}
```

Fonction de la classe Game 4

Par un fichier json

La fonction « initFromJson » est une méthode statique de la classe « Game » qui initialise une instance de la classe à partir d'un objet JSON. Elle prend un objet JSON en entrée, extrait les informations nécessaires pour créer des instances des classes « User » et « Backlog » en fonction du contenu du fichier, puis retourne une nouvelle instance de la classe « Game » avec ces données.

```
/**
 * Fonction pour initialiser une game à partir de l'objet d'un fichier JSON
 * @param {Object} json Objet JSON
 * @returns {Game}
 */
static initFromJson(json) {
    let users = []
    for (let user of json.newGame.users) {
        users.push(new User(user.name))
    }
    let backlogs = []
    for (let backlog of json.newGame.backlogs) {
        const finalRate = backlog.finalRate === undefined ? -2 : backlog.finalRate
        backlogs.push(new Backlog(backlog.title, backlog.description,
                                  Backlog.initRatesFromUsers(users), finalRate))
    }
    return new Game(json.newGame.name, json.newGame.mode, users, backlogs)
}
```

Fonction de la classe Game 5

Gestion du jeu

Le code concernant la gestion de la partie est situé dans le fichier « game.js » du répertoire « views ». Celui-ci reprend l'ensemble des fonctions détaillées plus tôt. Nous vous invitons à regarder le code, pour plus d'informations. L'identifiant de la partie est renvoyé entre chaque page par l'url, comme en témoigne le code suivant :

```
const gameId = new URLSearchParams(window.location.search).get('id');
const backlogId = new URLSearchParams(window.location.search).get('backlog');
const backlog = Backlog.initFromId(backlogId)
let currentIndex = -1
let cafe = []
console.log(backlog.rates)
displayUser(User.initFromId(backlog.rates[0].user))
const game = Game.initFromId(gameId)
```

Code game.js 1

Lorsqu'un tour de table entre les joueurs est terminé, l'utilisateur est renvoyé sur la page permettant de naviguer entre les backlogs (« game-resume.html »), comme en témoigne le code suivant. Enfin, si le tout n'est pas terminé, l'utilisateur reste sur la même page, et renvoie le nom de la prochaine personne devant soumettre le vote.

```
const gameId = new URLSearchParams(window.location.search).get('id');
const backlogId = new URLSearchParams(window.location.search).get('backlog');
const backlog = Backlog.initFromId(backlogId)
let currentIndex = -1
let cafe = []
console.log(backlog.rates)
displayUser(User.initFromId(backlog.rates[0].user))
const game = Game.initFromId(gameId)
```

Code game.js 2

Implémentation des tests unitaires

Lors de notre projet, de nombreux tests ont été implémentés, afin d'éviter la régression dans notre code. On proposera dans cette partie d'expliquer le fonctionnellement de quelques tests qui nous semblent important de détailler.

Tout d'abord, afin de mener à bien les tests unitaires, il a été nécessaire de reproduire le fonctionnement du LocalStorage. Cela passe par la création de la classe « LocalStorageMock », disponible dans les premières lignes du script « game_class.test.js ». En effet, il n'est pas possible d'utiliser les différentes fonctionnalités du LocalStorage dans les tests unitaires de Jest, ainsi on a créé un réplica du LocalStorage. De plus, plusieurs variables ont été créées, afin de simuler des objets de type « Game », « Backlog » et « User ».

La principale fonctionnalité à tester était l'importation d'un fichier json afin de redémarrer une partie déjà initialisée. Pour cela, on a défini une variable, nommé « json », simulant un fichier json conforme à notre application. Celui-ci est relativement simple et peu détaillé mais

convient pour les tests à réaliser. Pour les explications, je vous conseille de prendre appui du code ci-dessous :

```
describe("Tests unitaires classes", function () {
    it("should restart game from JSON", function () {
        const json = {
            game: {
                name: 'Test Game',
                mode: 'moyenne',
                users: [{ name: 'Alice', id: 'user1-id' }, { name: 'Bob', id: 'user2-id' }],
                backlogs: [
                    { title: 'Backlog 1', description: 'Description 1', rates: [], finalRate: 3,
                        id: 'backlog1-id', isFirstTurn: true },
                    { title: 'Backlog 2', description: 'Description 2', rates: [], finalRate: 5,
                        id: 'backlog2-id', isFirstTurn: false }
                ]
            }
        };

        const gameFromJson = Game.restartFromJson(json);

        expect(gameFromJson.name).toBe('Test Game');
        expect(gameFromJson.mode).toBe('moyenne');
        expect(gameFromJson.users.length).toBe(2);
        expect(gameFromJson.users[0].name).toBe('Alice');
        expect(gameFromJson.users[0].id).toBe('user1-id');
        expect(gameFromJson.users[1].name).toBe('Bob');
        expect(gameFromJson.users[1].id).toBe('user2-id');
        expect(gameFromJson.backlogs.length).toBe(2);
        expect(gameFromJson.backlogs[0].title).toBe('Backlog 1');
        expect(gameFromJson.backlogs[0].description).toBe('Description 1');
        expect(gameFromJson.backlogs[0].finalRate).toBe(3);
        expect(gameFromJson.backlogs[0].id).toBe('backlog1-id');
        expect(gameFromJson.backlogs[0].isFirstTurn).toBe(true);
        expect(gameFromJson.backlogs[1].title).toBe('Backlog 2');
        expect(gameFromJson.backlogs[1].description).toBe('Description 2');
        expect(gameFromJson.backlogs[1].finalRate).toBe(5);
        expect(gameFromJson.backlogs[1].id).toBe('backlog2-id');
        expect(gameFromJson.backlogs[1].isFirstTurn).toBe(false);
    });
```

Code Tests Unitaires 1

Dans ce test, on remarque que de nombreux éléments sont à vérifier : on utilise la fonction « restartFromJson » de la classe « Game », ayant un objet reproduit dans le fichier de test, puis on vérifie un par un les différents éléments du jeu généré. Ici, on peut citer plusieurs attributs soumis aux tests : le nom de la partie, le mode de jeu, les noms et id des différents utilisateurs, les attributs des backlogs, etc. Ce test, certes plus détaillé que les autres, permet une véritable simulation d'une partie initialisée à partir d'un fichier json et vérifie la valeur de chaque attribut. Un test, similaire à celui-ci, consiste à vérifier l'initialisation d'une partie grâce à un fichier json. Il ne sera pas détaillé dans le rapport, car l'unique différence consiste à initialiser une partie « gameFromJson » à partir de la fonction « initFromJson » de la classe « Game ».

Une fonctionnalité intéressante à tester est l'exportation d'une partie, également sous format json. Encore une fois, et comme en témoigne le script ci-dessous, on initialise une partie en créant une variable, conforme à la structure json demandée :

```
it("should export game to JSON", function () {
    const jsonExport = game.jsonExport();
    const expectedJson = JSON.stringify({
        game: {
            name: 'game1',
            mode: 'moyenne',
            users: [
                { name: 'user1', id: 'user1-id' },
                { name: 'user2', id: 'user2-id' }
            ],
            backlogs: [
                { id: 'backlog1-id', title: 'backlog1', description: 'description backlog 1',
                  rates: [], finalRate: -2, isFirstTurn: true },
                { id: 'backlog2-id', title: 'backlog2', description: 'description backlog 2',
                  rates: [], finalRate: -2, isFirstTurn: true }
            ]
        }
    });

    expect(jsonExport).toBe(expectedJson);
});
```

Code Tests Unitaires 2

Une variable « jsonExport » est créée à partir des variables ci-dessous, définies en début de script. On appelle ensuite la fonction « jsonExport » de la classe « Game » pour simuler une exportation de la partie créée.

```
//Définition d'objets de tests
const backlog1 = new Backlog("backlog1", "description backlog 1", [], -2, 'backlog1-id', true);
const backlog2 = new Backlog("backlog2", "description backlog 2", [], -2, 'backlog2-id', true);
const user1 = new User("user1", 'user1-id');
const user2 = new User("user2", 'user2-id');
const game = new Game("game1", "moyenne", [user1, user2], [backlog1, backlog2], 'game1-id');
```

Variables Tests Unitaires 1

Une variable « expectedJson » est ensuite instanciée et reprend les informations des variables ci-dessus mais cette fois-ci en format json. Enfin, on vérifie que le fichier générée avec la fonction d'importation est égal au résultat attendu, en fonction des variables définies.

Bien évidémment, d'autres tests unitaires ont été implémentés, afin de couvrir la majorité des fonctionnalités développés. L'ensemble des tests est disponible dans le fichier « game_class.test.js », disponible dans le répertoire « data ».

Intégration continue

Préambule

L'intégration continue, pour notre projet, a été réalisé par avec les différents outils intégrés de GitHub, permettant de faciliter le déploiement. Dans cette section, on expliquera plus en détail

le code permettant l'intégration continue. Celui-ci est disponible dans le répertoire .github/workflows de notre repository. Analysons plus en détail la structure de code ci-dessous, permettant l'intégration continue :

```yaml
name: Test_Unitaires_JS

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: windows-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '22.11.0'

      - name: Install npm
        run: npm install -g npm@10.9.0

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test

      - name: Generate documentation
        run: npm run generate-docs
```

```yaml
- name: Deploy to GitHub Pages
  uses: peaceiris/actions-gh-pages@v3
  with:
    github_token: ${{ secrets.GITHUB_TOKEN }}
    publish_dir: ./docs
```

Code du fichier .github/workflows/.yaml*

La première chose à constater est la liste des événements permettant l'intégration continue. Dans notre cas, on a décidé de la générer pour chaque « push » ou « pull_request » effectué dans notre dépôt GitHub. On aurait pu se contenter uniquement de l'événement « push », mais on a trouvé tout de même pertinent l'appel de l'événement « pull_request », celui-ci étant très utilisé dans les projets à plus grande ampleur. Cet événement n'a en effet pas pu être sollicité lors de notre développement et cela pour plusieurs raisons. La raison principale repose sur le nombre de contributeurs du projet, celui-ci étant très faible, car le travail se faisait par binôme. De ce fait, l'intégralité du processus de conception et d'intégration du « Planning Poker » s'est fait en équipe, évitant les phases de validation de code. On peut relier cette pratique au « Pair Programming », résultant de la méthode XP (eXtreme Programming) abordé en cours.

Les déclencheurs de l'intégration continue sont portés uniquement sur la branche « main » de nôtre projet. En effet, la mise en pratique du « Pair Programming » ne nécessitait pas l'utilisation de sous branches, comme des branches de développement de fonctionnalités ou de releases. Cela s'explique encore une fois par la taille de l'équipe et du fait toujours coder par paires. L'ensemble des commits réalisés étaient donc validés et fonctionnels, ce qui facilite la gestion des branches.

Ayant désormais explicité nos choix, on présentera ci-dessous l'automatisation des tests unitaires et la génération de la documentation, pour chaque « push » ou « pull_request »

Automatisation des tâches

L'ensemble des tâches effectuées seront exécutées à partir d'un seul job. Chaque action (steps) est nommée en fonction de son utilité, par soucis de simplicité. Tout d'abord, concernant l'environnement (runner), on a favorisé l'utilisation de Windows, celui-ci étant par défaut sur nos machines.

Il est nécessaire, de configurer l'environnement de travail pour les tests et la documentation, et ceci de façon automatique, pour répondre au changement dans le code ou autre modification. Pour cela, analysons brièvement chaque step plus en détail.

Dans un premier temps, récupérer le code courant est essentiel pour le bon déroulement de l'automatisation. L'étape « Checkout code » est utilisée pour récupérer le code source de notre dépôt GitHub dans l'environnement de l'action. Cela permet aux autres étapes du workflow d'accéder aux fichiers du projet pour exécuter des tests, générer de la documentation, etc. A partir de notre code récupéré, on met en place notre environnement de travail, ici gérer par Node.Js. Le fichier d'automatisation doit donc automatiser pour chaque changement dans notre code la création d'un nouvel environnement, prêt à l'emploi pour les tests et la documentation. L'étape « Set up Node.js » configure l'environnement Node.js pour le workflow. Cela inclut l'installation de la version spécifiée de Node.js, ce qui permet d'exécuter des commandes

Node.js et npm dans les étapes suivantes du workflow. Ensuite, l'étape « Install dependencies » installe les dépendances du projet en utilisant npm. Cela garantit que toutes les bibliothèques et modules nécessaires sont disponibles pour les étapes suivantes du workflow.

Maintenant que notre environnement est correctement automatisé, on peut dès à présent effectuer l'automatisation des tests et de la documentation. les étapes « Run tests » et « Generate documentation » sont cruciales pour vérifier la qualité du code et générer la documentation du projet. Voici une explication détaillée de ce que fait chaque étape.

Tests Unitaires

L'étape « Run tests » exécute les tests unitaires pour vérifier que le code fonctionne correctement. Ceux-ci sont définis grâce à un framework de test, qui est Jest dans notre cas. L'étape exécute ainsi la commande npm test, définie dans le fichier « package.json », définit plus bas. Jest lit uniquement les fichiers de test, comprenant l'extension « .test.js » et exécute les tests présents dans les fichiers trouvés.

Génération de la documentation

L'étape « Generate documentation » génère la documentation du projet en utilisant un outil, qui est JSDoc dans notre cas. Cela inclut la génération de fichiers HTML ou Markdown décrivant les classes, méthodes, et autres éléments du code. L'étape exécute la commande npm run generate-docs, qui est définie dans le fichier « package.json » pour générer la documentation en utilisant JSDoc. Les fichiers de documentation sont ensuite stockées dans la branche « gh-pages » avec l'étape « Deploy to GitHub Pages ». Cependant nous n'avons pas eu le temps de comprendre comment fonctionner les GitHub pages, donc ces documents sont dans cette branche mais ne sont pas déployer. Ces fichiers peuvent être consultés pour obtenir des informations détaillées sur le projet.

Pour plus d'informations, veuillez trouver ci-dessous le script du fichier « package.json » :

```
{
  "type": "module",
  "scripts": {
    "test": "jest",
    "test:coverage": "jest --coverage",
    "generate-docs": "npx jsdoc -c ./jsdoc.json"
  },
```

Code de package.json pour l'IC

Conclusion

Ce projet a été très formateur concernant les méthodes de conception d'un projet informatique. On peut apporter quelques critiques sur notre projet, permettant à l'avenir de l'améliorer. Tout d'abord, il est clair que le style de CSS est à revoir, afin de donner un aspect unique à l'application. En effet, par faute de temps, une partie du fichier CSS a été réalisé à l'aide d'une IA. Enfin, une version jouable en ligne est aussi souhaitable, mais a été trop complexe a réalisé

pour ce projet. Néanmoins, nous sommes satisfaits du travail réalisé pour le temps qui nous a été confié.