

Université Lumière Lyon 2

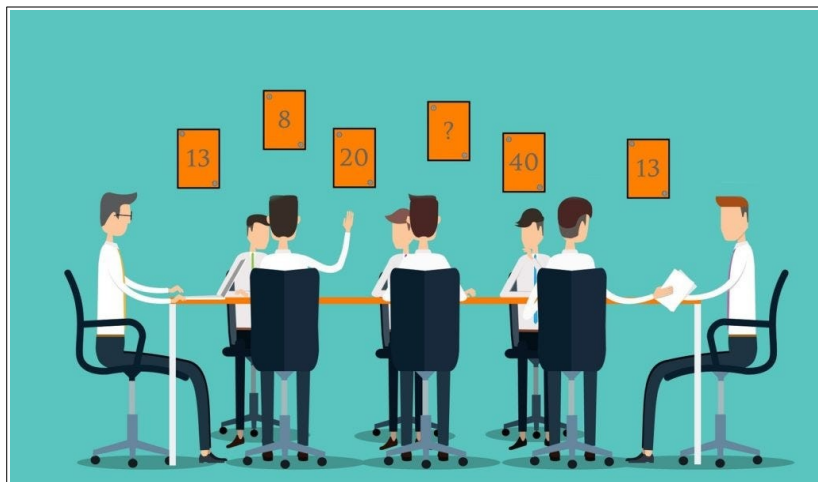


RAPPORT DE PROJET  
CONCEPTION AGILE DE PROJETS INFORMATIQUES

---

Projet : application de “planning poker”

---



Flavien Gonin

N°étudiant : 2235811

Guillaume Hostache

N°étudiant : 2236156

# I - Présentation générale

L'application développée est une application de planning poker. Le but d'un planning poker est d'estimer à plusieurs les difficultés de plusieurs tâches pour le backlog d'un projet. Dans le cas de cette application, le backlog est un fichier au format .json avec le nom du projet, toutes les tâches a réalisé ainsi que le détail de ces tâches. À partir de ce fichier et de l'application, un groupe de 2 à 4 personnes peuvent faire une partie de planning poker en local. Ils peuvent choisir entre plusieurs modes de jeu (strict ou moyenne). Dans le mode strict, il faut que tous les joueurs soient unanimes avant de passer à la tâche suivante. Dans le mode moyenne, il faut soit être unanimes lors du premier tour où alors au deuxième tour, l'application fera une moyenne des cartes sélectionner avant de passer à la tâche suivante.

Lorsque la partie est terminée, l'application produit un fichier de sortie "backlog\_sauvegarde.json" qui contient toutes les tâches avec les difficultés attribuées.

L'application permet aussi de faire des pauses, pour cela, tous les joueurs doivent choisir la carte café. Cela permettra de sauvegarder l'état actuel de la partie dans le fichier "backlog\_sauvegarde.json". Si c'est le cas, l'équipe pourra directement reprendre la partie, là où elle était arrêtée, avec ce fichier.

## II – Présentation du code

### 1. Choix technologiques, architectures

De base, on avait choisi de réaliser ce projet avec Java et la librairie JavaFx, mais nous avons rencontrés des problèmes pour lancer le projet (impossible de faire fonctionner JavaFx) donc nous sommes parties sur l'utilisation des technologies Web "natif". Après réflexion, les technologies web (Html, Css, JavaScript...) étaient plus facile à utiliser pour nous et permettent plus simplement de mettre en place à l'avenir une version multi-joueurs de ce logiciel.

Au niveau de l'architecture, nous avons utilisé quelque chose de très basique avec nodejs, et pas de framework car aucun de nous deux avaient de solide connaissance en framework web.

## 2. Les patrons de conception

Tout d'abord, nous avons utilisé un itérateur (patron de comportement). Celui-ci permet de lire la liste de tâches du fichier json au fur et à mesure de l'avancer des tâches. L'avantage de celui-ci est de ne pas avoir à se rappeler à chaque fois l'index de la tâche courante mais juste de consulter l'itérateur afin d'avoir les informations demandées.

```
22
23
24 /**
25  * Itérateur de la liste de tâche d'un fichier json
26  * @param fichierJson - fichier json contenant les tâches
27  * @returns {{next: (function(): ({value: *, done: boolean}})}}
28 */
29 function listeTaches(fichierJson) {
30     let indexTache = 0;
31     return {
32         next: function () {
33             let tache;
34             if (indexTache < fichierJson['liste_tache'].length) {
35                 tache = (value: fichierJson['liste_tache'][indexTache], done: false);
36                 indexTache++;
37                 return tache;
38             }
39             return {value: "", done: true};
40         };
41     };
42 }
```

Ci-contre, le code utiliser pour l'itérateur. Celui-ci renvoie une fonction `next()` qui va à chaque nouvelle appel, recalculer les informations demandées.

Toutes les informations seront dans le `next().value`. On utilisera `next().done` pour savoir si la fin de la liste à été atteinte.

Pour le deuxième patron, nous avons utilisé un patron de structure : l'adaptateur. Dans notre cas, l'on s'en sert de la manière suivante : en entrée on a toutes nos données en vrac (saisies formulaires, fichier Json [avec deux formats différents]) et en sortie, les données sont adaptés (si besoin), validés et envoyer dans le session storage afin que l'on puisse les récupérer et les utiliser facilement par la suite. Nous représentons notre adaptateur avec une classe. On crée une instance de cette classe lorsque l'on clique sur le bouton qui valide une partie du formulaire. Puis on appelle `adapterFichierJson` qui traite le fichier backlog fourni. On utilise la mécanique de promesse (très complexe et importante en web).

```
253 function validerFormulaire(option) {
254     let saisie = (option === 0) ? get('jsonFileLancer') : get('jsonFileReprendre');
255     if (saisie.files.length !== 0) {
256         let adaptateur = new Adaptateur(option);
257         // réception d'une promesse
258         adaptateur.adapterFichierJson(option)
259             .then(() => {
260                 // tout se passe bien, aucune erreur reçue
261                 window.location.href = 'jeux.html';
262             })
263             .catch((erreur) => {
264                 // On a reçu une erreur, il y a eu un soucis
265                 alert("Erreur - Le fichier fourni est incompatible avec l'application ! ");
266                 console.error("Erreur (" + erreur + ") - On ne peux pas traiter le fichier Json sélectionné !");
267                 nettoyerMenu(option);
268             });
269     }
270 }
```

```

js main.js M X
src > js main.js > Adaptateur
1  /** Classe Adaptateur qui permet d'adapter les données
2  * En entrée, on aura toutes les données 'en vrac', ens
3  * le 'Session Storage' ainsi, dans la page 'jeux.html'
4  * souhaite.
5  */
6  class Adaptateur {
7  /** Constructeur de la classe Adaptateur
8  * @param {0 | 1} option - Type de partie à lancer
9  */
10 constructor(option) {
11     clearStorage();
12     if (option === 0) {
13         this.adapterMode();
14         this.adapterNbJoueurs();
15         this.adapterNomJoueurs();
16     }
17 }

```

La classe ne possède aucun attribut, son constructeur prend en paramètre option qui lui permet de savoir comment adapter les données en fonction de la partie que l'on veut lancer (nouvelle ou ancienne partie).

Dans le constructeur, on s'occupe directement de charger les petites informations dans le cas d'une nouvelle partie, sinon, c'est dans adapterFichierJson(option) que l'on s'occupe du reste.

```

24 adapterFichierJson(option) {
25     return new Promise((resolve, reject) => {
26         let base = (option === 0) ? get('jsonFileLancer') : get('jsonFileReprendre');
27
28         if (base.files.length > 0) {
29             let fichier = base.files[0];
30             let lecteur = new FileReader();
31
32             lecteur.onload = function (evt) {
33                 try {
34                     let fichierJson = JSON.parse(evt.target.result);
35
36                     if (option === 1 && !('mode' in fichierJson) && ('nb_joueurs' in fichierJson) && ('liste_joueurs' in fichierJson)) {
37                         console.error("Erreur - Vous tentez de reprendre une partie avec un fichier qui ne contient pas assez d'informations !");
38                         reject(-1);
39                     }
40
41                     if ('liste_tache' in fichierJson) {
42                         if (option === 1) {
43                             let position = 0;
44                             let liste = fichierJson['liste_tache'];
45
46                             for (let i = 0; i < liste.length; i++) {
47                                 let maTache = fichierJson['liste_tache'][i];
48                                 if (maTache['difficulte'] !== "") {
49                                     position += 1;
50                                 }
51                             }
52
53                             saveData('position', (position !== 0) ? position : null);
54                         }
55
56                         saveData('fichierJson', JSON.stringify(fichierJson));
57                         resolve(0);
58                     } else {
59                         console.error("Erreur - Le fichier JSON ne contient pas la clé 'liste_tache' !");
60                         reject(-1);
61                     }
62                 } catch (error) {
63                     console.error("Erreur - Le fichier n'est pas un JSON valide !");
64                     reject(-1);
65                 }
66             };
67         }
68
69         lecteur.readAsText(fichier);
70     } else {
71         console.error("Erreur - Aucun fichier n'est sélectionné !");
72         reject(-1);
73     }
74 }
75 }

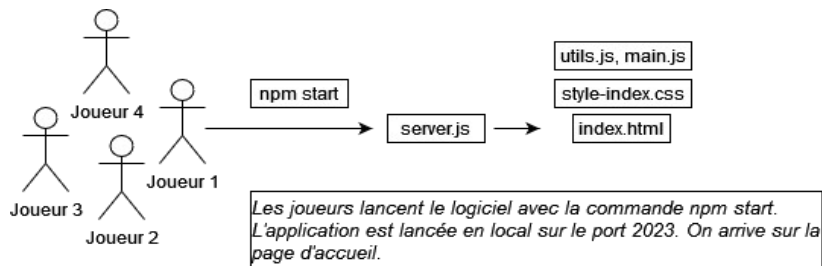
```

Dans cette méthode, on récupère la saisie, première vérification pour être sûr que l'on a bien quelque chose. Ensuite si c'est le cas, on lance la lecture du fichier. On capture les erreurs possibles, on regarde s'il est impossible de convertir le fichier en contenu Json (format invalide) et après, si dans le contenu json du fichier, il y a bien une clé 'liste\_tache' (a minima). On retourne 0 si tout est bon et -1 dans le cas contraire.

Nous avons cherché un troisième patron de conception, dans l'idéal, nous cherchions un patron de création mais nous n'en avons pas trouvé un qui aurait une réelle utilité dans notre situation.

### 3. Quelques explications de code

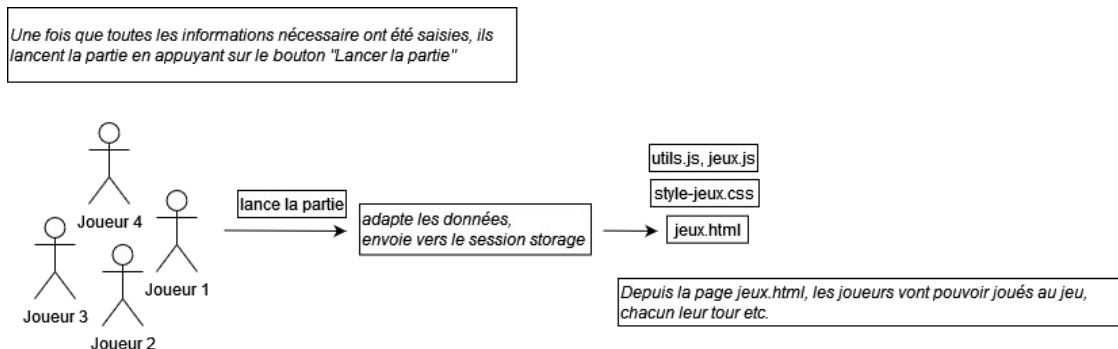
Tout d'abord, quelques explications sur le fonctionnement.



L'équipe saisie les informations qu'elle veut pour lancer une nouvelle partie ou reprendre une ancienne partie. Dans les deux cas, ils faudra fournir un backlog qui devra respecté les formats suivants :

```
1 {
2   "nom_projet": "Construction d'un Lego City",
3   "liste_tache": [
4     {
5       "nom_tache": "Préparer l'espace pour construire le lego.",
6       "details": "Il faut préparer un espace sur une table."
7     },
8     {
9       "nom_tache": "Ouvrir la boîte et trier les pièces de lego",
10      "details": "D'abord, ouvrir la boîte puis tout les différents sachets, trier les pièces de façon à faciliter la construction."
11    },
12    {
13      "nom_tache": "Vérification de l'état et du nombre de pièce.",
14      "details": "Ouvrir la notice vers les dernières pages, contrôler le bon nombre de pièce."
15    },
16    {
17      "nom_tache": "Première lecture de la notice.",
18      "details": "On va lire une première fois la notice, ça permet d'avoir un aperçu de ce qui nous attend."
19    },
20    {
21      "nom_tache": "Construction du lego",
22      "details": "Il faut faire le lego, lire chaque page et suivre les instructions"
23    },
24    {
25      "nom_tache": "Prendre plaisir avec son lego",
26      "details": "S'amuser, le mettre dans une belle vitrine avec toute sa collection"
27    }
28  ]
29 }
```

```
1 {
2   "mode": "moyenne",
3   "nb_joueurs": 3,
4   "liste_joueurs": [
5     "Mendoza",
6     "Sancho",
7     "Pedro"
8   ],
9   "nom_projet": "Découverte des mystérieuses cités d'or",
10  "liste_tache": [
11    {
12      "nom_tache": "Découvrir la première cité d'or",
13      "details": "C'est le début de l'aventure ...",
14      "difficulte": "3"
15    },
16    {
17      "nom_tache": "Faire face à Zarès et s'enfuir",
18      "details": "Zarès veut aussi les cités d'or, il nous attaque par surprise en Italie, il faut s'enfuir !",
19      "difficulte": "20"
20    },
21    {
22      "nom_tache": "En route vers les autres cités d'or",
23      "details": "On se met en route vers l'Amérique et le Japon pour découvrir les autres cités d'or",
24      "difficulte": "13"
25    },
26    {
27      "nom_tache": "Zarès nous suit de près, mais qui est-il ?",
28      "details": "Il faut démasquer Zarès et le vaincre",
29      "difficulte": ""
30    },
31    {
32      "nom_tache": "Direction la Chine pour une autre cité d'or",
33      "details": "Prochaine piste, en Chine ! Mais cette cité d'or à l'air plus compliquée à trouver",
34      "difficulte": ""
35    },
36    {
37      "nom_tache": "Affronter Zarès",
38      "details": "Arrivé aux portes de la 4e cité d'or, Zarès nous attend ...",
39      "difficulte": ""
40    }
41  ]
42 }
```



Désormais, le “jeu” va commencer. Lors d’une partie, un objet `maPartie` va être créé et contiendra toutes les informations utiles à la partie, c’est-à-dire, le nombre de joueurs ainsi que leurs noms et le mode de jeu en plus de la liste de tâches fournie avec le fichier `Json`. Lors de l’initialisation de la partie, on va afficher toutes les informations utiles à l’écran. Parmi-elles, se trouvent le nom du projet, le titre de la tâche et les détails de celle-ci. Lors du chargement, on va récupérer les informations qui se trouvent dans le session storage. En plus de cela, lors d’une reprise de la partie, on va aussi commencer à re-sauvegarder les tâches qui ont déjà été définies. Après tout cela, on peut commencer la partie.

Le déroulement d’un tour dépend grandement des règles, mais consistera toujours en 3 phases. La première pendant laquelle chaque joueur choisira sa carte chacun à son tour. La deuxième où les cartes choisies par les différents joueurs sont affichées, leur laissant le temps de parler. Et enfin, la troisième, où l’application agit selon les résultats (changer de tâche, recommencer le tour ou pause café). Lors de la phase de choix, les joueurs n’ont que 30 secondes pour se décider. Si aucune carte n’est sélectionnée avant la fin du chronomètre, une carte “?” est jouée à la place. Lors de la deuxième phase, il y a aussi un chronomètre mais, cette fois-ci de 60 secondes.

Lorsque la fin du fichier de tâche est atteinte ou que la pause café est choisie, l’application sauvegarde toutes les informations de la partie, soit les difficultés choisies et les participants ainsi que le mode. Après un retour sur l’écran d’accueil, le fichier est téléchargé et le jeu prend fin. Si la fin est atteinte par la pause café, on remplit les difficultés avec des chaînes de caractères vides.

## 4. Tests unitaires

Lors du développement de cette application, des tests unitaires ont été créés afin de s'assurer du bon fonctionnement de l'application. La plupart des tests permettent de tester le bon fonctionnement de la page d'accueil. En outre, les fonctionnalités testées sont les suivantes :

- l'affichage des règles / des menus / des règles dans le menu
- un test permettant de tester la bonne installation des tests
- la suppression d'information dans les menus
- affichage des zones de saisie en fonction du nombre de joueurs

Nous avons aussi des tests pour la partie “jeu”, notamment sur l'itérateur. Ils nous permettent de :

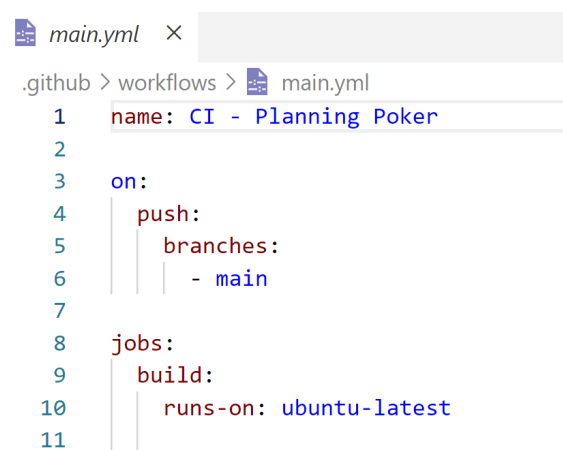
- vérifier la conformité des informations
- vérifier le bon nombre d'objets dans l'itérateur
- vérifier la condition d'arrêt de l'itérateur

Cependant, la rédaction des tests a été l'une des plus grandes difficultés rencontrées dans le projet et c'est pour cela que notre couverture de test est loin d'être “robuste” (En complément, dans le répertoire ressources, nous avons fait des tests fonctionnels en créant quelques exemples de backlog différents pour une nouvelle partie comme une ancienne partie ...).

## III - Mise en place de l'Intégration Continue

Nous avons travaillé en utilisant le gestionnaire de source Git et la plateforme web GitHub et ses services intégrés. Nous avons mis en place l'intégration continue relativement tôt dans le développement du projet de la manière suivante.

Notre intégration continue s'appelle “CI – Planning Poker”. Le workflow est déclenché à chaque push sur la branche principale “main”.



```
1 name: CI - Planning Poker
2
3 on:
4   push:
5     branches:
6       - main
7
8 jobs:
9   build:
10     runs-on: ubuntu-latest
11
```

Le workflow contient un seul travail (jobs) qui est configuré pour s'exécuter sur la dernière version d'Ubuntu. Ce boulot contient plusieurs étapes.

D'abord, on récupère le code du dépôt et on installe nodejs sur l'environnement de travail virtuel (on précise que l'on veut la version 16)

```
11
12 steps:
13 - name: Checkout Repository
14   uses: actions/checkout@v2
15
16 - name: Setup Node.js
17   uses: actions/setup-node@v3
18   with:
19     node-version: 16
20
21 - name: Install Dependencies
22   run: npm install
23
24 - name: Run Tests
25   run: npm test
26
27 - name: Generate Documentation
28   run: npm run docs
29
30 - name: Deploy to GitHub Pages
31   uses: peaceiris/actions-gh-pages@v3
32   with:
33     github_token: ${ secrets.GITHUB_TOKEN }
34     publish_dir: docs/
35
36
```

Ensuite, on installe les dépendances liées au projet avec la commande “npm install”. On vérifie qu’il n’y a aucun problème en lançant les tests unitaires du projet avec la commande “npm test”.

Enfin, on génère la documentation avec la commande “npm run docs” puis on se charge du déploiement du projet sur GitHub Pages.

Dans notre cas, le déploiement est réalisé à l'aide de l'action “peaceiris/actions-gh-pages@v3”. Cette action utilise le jeton d'accès de GitHub “secrets.GITHUB\_TOKEN” et précise le dossier que l'on va publier, ici, le dossier “docs/” contenant la documentation du projet.