

INSTITUT DE LA COMMUNICATION - UNIVERSITÉ LUMIÈRE LYON 2

RAPPORT DE PROJET

## Sujet : Réalisation d'une application de planning poker locale en Python

---

*Auteurs :*

Enzo MARTINEZ

Angelo LAMURE-FONTANINI

*Référent :*

M. Valentin LACHAND-PASCAL

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Déroulé d'une partie</b>	<b>3</b>
2.1	Page d'accueil . . . . .	3
2.2	Pre-Prog du planning poker . . . . .	4
2.3	Déroulé d'une partie . . . . .	5
<b>3</b>	<b>Choix techniques</b>	<b>7</b>
3.1	Architecture . . . . .	7
3.2	Langage . . . . .	8
3.3	Classes . . . . .	8
<b>4</b>	<b>Design pattern</b>	<b>9</b>
4.1	Singleton . . . . .	9
4.2	Factory . . . . .	9
4.3	Composite . . . . .	10
<b>5</b>	<b>Intégration continue</b>	<b>11</b>
5.1	Tests Unitaires . . . . .	11
5.2	Génération de la documentation . . . . .	11
5.3	GitFlow . . . . .	11

# 1 Introduction

---

Ce rapport fait office de présentation du projet réalisé dans le cadre de la matière "Conception agile de projet".

Il sert de support à la présentation de notre projet dans le cadre de l'évaluation de ladite matière. Il contient, un exemple d'utilisation de l'application, une brève présentation de l'architecture du projet ainsi que des choix réalisés au cours de son développement (choix langue, patrons de conceptions ...).

En plus de cela, une section est dédiée à l'intégration continue réalisée au cours du projet (Automatisation des tests unitaires, automatisation de la doc, gitflow).

Le présent rapport a donc pour but de venir justifier et expliquer nos choix techniques au fur et à mesure du développement de l'application de Planning Poker ainsi que mettre en valeur le travail effectué.

## 2 Déroulé d'une partie

---

Cette section a pour but de montrer les fonctionnalités implémentées dans notre application et comment y accéder afin que l'application soit plus accessible aux utilisateurs.

### 2.1 Page d'accueil

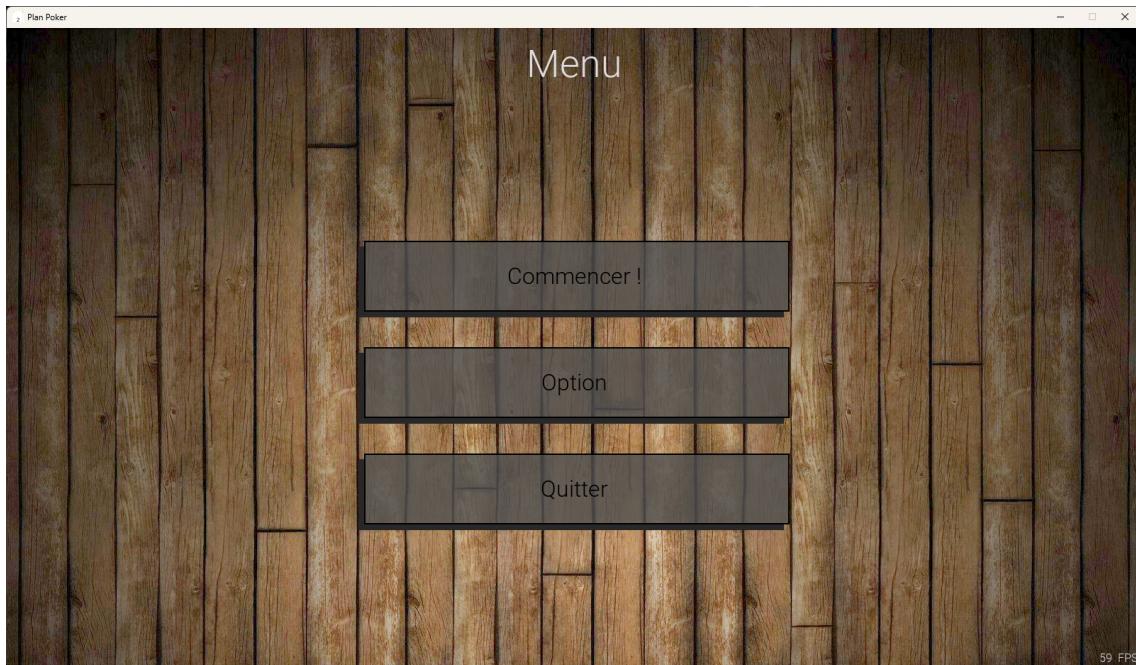


FIGURE 1 – Page d'accueil de l'application

Quand l'utilisateur lance l'application, il arrive directement sur la Figure 1 où il a la possibilité de soit, lancer le planning poker, soit, changer les options, soit, de quitter l'application.

L'onglet " Option " permet de personnaliser l'expérience de l'utilisateur. Il peut par exemple décider de limiter les FPS, où encore de changer les paramètres audio.

Dans le scénario nominale d'utilisation de l'application, l'utilisateur va alors cliquer sur le bouton "Commencer !" et va alors être envoyé sur la page suivante, Figure 2.

## 2.2 Pre-Prog du planning poker

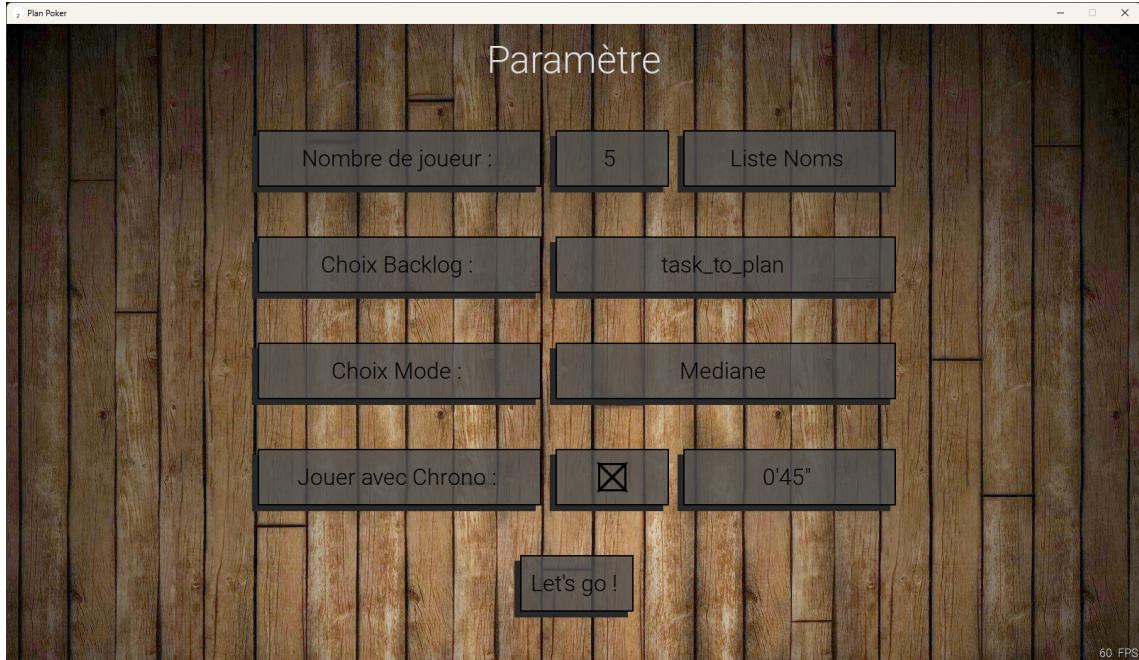


FIGURE 2 – Page de configuration du planning poker

Sur cette page, l'utilisateur a plusieurs options à configurer avant de lancer le planning poker.

Dans l'ordre, nous avons :

- Le choix du nombre de joueurs ainsi que leurs noms avec une page associée à la configuration de ceux-ci.
- Le choix du backlog sur lequel va se porter la partie / réunion de planning poker.
- Le choix du mode (unanimité, moyenne, médiane ...)
- Le choix de mettre un "chrono" à la partie, c'est à dire, qu'il y ait un temps limité au choix des coûts des tâches. On peut cocher la case (pour activer le chrono) et changer la durée du chrono. Ici, le temps est fixé entre 10s et 5 min.

Il y a quelques fenêtres qui sont encore présentes sur cette page mais qui sont accessibles seulement dans des cas spécifiques comme par exemple, si les personnes présentes lors de la réunion décident de sélectionner un backlog déjà entièrement rempli, alors une fenêtre va s'afficher leur demandant si elles veulent recommencer le backlog depuis le début ou non.

Une fois les paramètres sélectionnés, toujours dans le scénario nominal précédent, l'utilisateur va alors lancer la réunion en cliquant sur le bouton "Let's go!"

## 2.3 Déroulé d'une partie

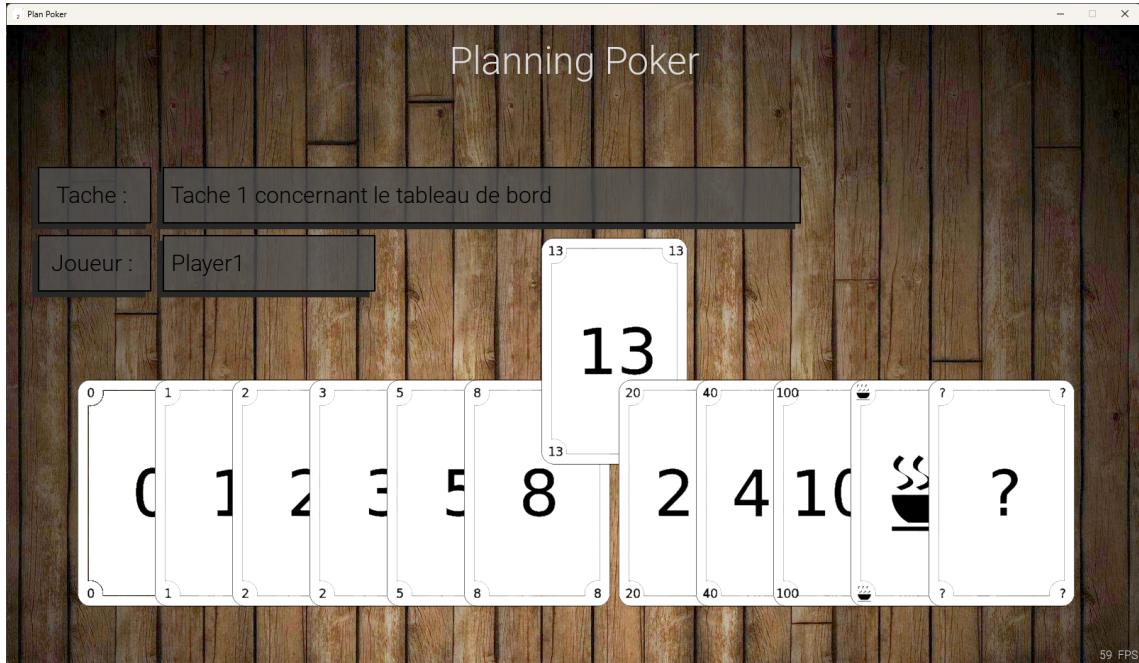


FIGURE 3 – Exemple d'une partie de planning poker

Une fois que le planning poker est lancé, on se retrouve sur cette page avec le choix des cartes, la tâche en cours de vote ainsi que le joueur qui sélectionne le coût qui lui convient. Il est à préciser que sur la Figure 3, il n'y a pas la présence du timer car, l'option n'a pas été sélectionnée.

Chaque joueur choisit donc la carte qui lui convient et, selon le mode de vote, les membres de la réunion, arrivent sur cette page.



FIGURE 4 – Résumé des votes

Il est à préciser que si au moins un joueur choisit l'option café, alors l'option de sauvegarder le backlog est alors proposée. Pour décrire les choix Figure 4, on voit en couleur verte et rouge, les choix les plus extrêmes avec en rouge le plus élevé et en vert le plus bas. On voit aussi que deux utilisateurs ont choisi l'option "?" et finalement les choix intermédiaires qui sont en gris.

Suite à cette étape de résumé des choix des membres de la réunion, selon le mode de vote, s'ensuit une phase d'explication. Les joueurs ayant voté les choix les plus extrêmes ont alors la possibilité d'écrire un paragraphe (255 caractères) sur les raisons de leurs choix.

Une fois cette étape passée, on retourne sur la page "résumé", mais cette fois, lorsque l'on survole le pseudo d'un des 2 joueurs, alors, l'explication de son choix s'affiche sur l'écran comme montré ci-contre.



FIGURE 5 – Explication du "Player 2"

Cette fonctionnalité représente l'option de "chat" demandée par notre encadrant. Nous avons choisi de la représenter de cette manière plutôt que comme un chat classique / discussion type application messenger car notre application prend déjà place en local.

Suite à cette page de résumé, on retourne au choix des coûts de la tâche courante et ainsi de suite tant que les choix des membres de la partie ne font pas consensus par rapport au mode de jeu sélectionné.

Il est aussi à préciser que peut importe quand l'application est fermée, le backlog est alors enregistré en mémoire de l'ordinateur ainsi que les options de backlog.

### 3 Choix techniques

#### 3.1 Architecture

Nous avons décidé d'organiser notre projet en plusieurs fichiers python ainsi que plusieurs dossiers. Pour mieux comprendre les liens entre les fichiers, nous avons décidé de réaliser un diagramme de classe simplifié, sans les fonctions, essentiellement pour montrer les liens entre les fichiers de notre projet.

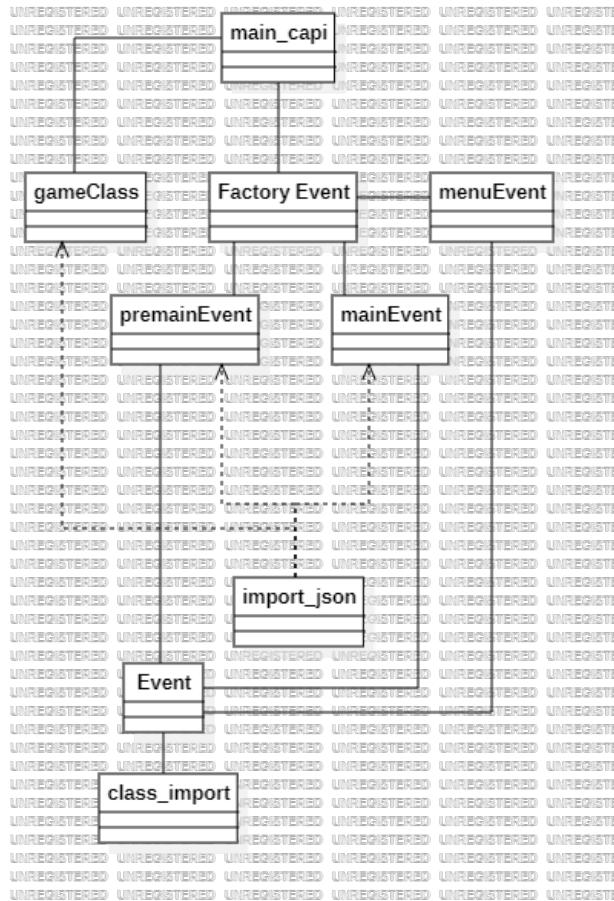


FIGURE 6 – Pseudo diagramme de classe

Il est à préciser que le fichier `import_json` n'est pas une classe mais est importé dans plusieurs fichiers pour permettre l'utilisation des fonctions incluses dans ledit fichier. Il faut savoir qu'en python, réaliser l'import d'un fichier externe équivaut à écrire le code du fichier directement dans le même fichier. Voilà pourquoi nous avons préféré différencier les liens des classes "normales" de celles du fichier `import_json`.

Comme on peut le voir, nous avons organisé notre projet au centre du fichier `main_capi` qui exécute la boucle principale du code.

`main_capi` alterne entre 3 pages `MenuEvent`, `MainEvent` et `PremainEvent` qui sont organisées sous la forme d'un patron de conception dont nous parlerons dans la catégorie associée (Factory Pattern).

Chaque Event va permettre de gérer les différents interface soumis à l'utilisateur. Pour être plus précis, quand l'utilisateur démarre l'application de planning Poker, il va arriver directement dans l'accueil, qui est géré par la classe `MenuEvent`. Ensuite, si l'utilisateur décide de passer au planning poker, il cliquera sur "Commencer!", et l'on va maintenant passer aux paramètres de partie, et c'est donc la classe `PremainEvent` qui prendra le relais.

Chacun de ces trois Event ont des sous-event, c'est-à-dire des classes qui permettront de gérer des sous-événements. Par exemple, dans le `PremainEvent` où il est possible de changer le pseudo des joueurs, c'est un sous-event appelé `SetNameEvent` qui va s'en occuper.

## 3.2 Langage

Pour ce projet, nous avons opté comme langage de programmation pour le python. Ce langage est plutôt facile de prise en main tout en assurant un développement fluide avec les connaissances que nous avions tous les deux du-dit langage. En effet, Angelo avait déjà une grande connaissance de la bibliothèque "pygame", que nous avons utilisé lors de ce projet. Cette bibliothèque permet un développement facile et intuitif d'interface graphique et plus précisément de jeux vidéos en Python.

En plus de cela, Python est un langage objet et donc permet la mise en place de design pattern demandés par notre enseignant.

## 3.3 Classes

Nous avons décidé de séparer les différentes classes en différents fichiers hormis l'exception de `import_json` qui n'est pas une classe à part entière ainsi que `class_import` qui contient plusieurs classes en fonction de ce que l'on veut importer dans nos pages.

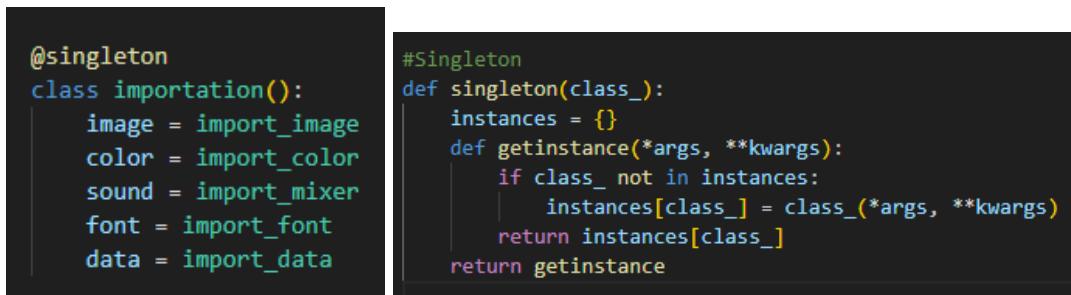
## 4 Design pattern

Au cours de ce projet, nous avons eu l'occasion de mettre en place nos connaissances sur les patrons de conception vus en cours. Nous avons décidé d'implémenter trois d'entre eux, un Singleton Pattern, un Factory (Fabrique) pattern ainsi qu'un patron Composite.

### 4.1 Singleton

Le premier pattern que nous avons décidé de mettre en place est un singleton pattern. Ce patern permet de garantir l'unicité d'une classe lors de sa création.

Nous l'utilisons dans notre projet sur la classe importation qui se situe dans class\_import car cette dernière est bien sollicitée une et unique fois par event mais event, lui, est sollicité un grand nombre de fois dans notre projet. Il nous a donc fallu garantir l'unicité de "importation" pour éviter la surcharge mémoire.



```
@singleton
class importation():
    image = import_image
    color = import_color
    sound = import_mixer
    font = import_font
    data = import_data

#Singleton
def singleton(class_):
    instances = {}
    def getinstance(*args, **kwargs):
        if class_ not in instances:
            instances[class_] = class_(*args, **kwargs)
        return instances[class_]
    return getinstance
```

FIGURE 7 – Singleton sur la classe importation

Comme vous pouvez le voir plus haut, nous rajoutons sur la classe "importation" un décorateur @singleton, de la même manière que l'on ajoute une annotation en java.

Ce décorateur fait référence à la méthode "singleton" définie plus haut. On peut définir de plusieurs manières un singleton en Python et cette manière prend exemple sur les patrons de décoration (décorateur), cf : lien de l'exemple si dessus.

Ce patron de conception fait donc référence à deux patrons en un, le patron singleton ainsi que le patron décorateur, mais nous préférons le compter comme un seul patron, de part surtout le contexte d'évaluation et la demande d'avoir 3 patrons de conception au sein de notre projet.

Le gros problème de cette méthode est que "importation" est alors traité comme une fonction et donc, n'autorise pas l'appel de méthode sur lui même, comme le ferait un objet classique.

Dans notre cas, cela ne pose pas de problème étant donné que "importation" sert à, comme son nom l'indique, importer des variables / assets utiles pour le projet (taille des boutons, textes, etc).

### 4.2 Factory

Le deuxième design pattern que nous avons implémenté est un Factory pattern ou encore une Fabrique.

Lors de l'utilisation de notre application, comme vu sur le diagramme de classe de la section "architecture", notre projet va afficher certaines pages en fonction d'un contexte. Étant donné que ce contexte peut changer au fur et à mesure du projet, reprise de celui-ci, du temps, le nombre de pages va alors être amené à changer. Pour cela, il nous a paru évident de centraliser la création de ces pages dans une même classe "factoryEvent".

Le gros avantage d'un factory pattern est donc, comme expliqué au dessus de la figure 4.2, de permettre un ajout plus facile de pages à notre application si elle vient à être modifiée dans le futur.

Nous centralisons les informations dans "FactoryEvent" pour plus de clarté et nous permettons une meilleure modularité (ajout / suppression de page) de notre application.

```

from menuEvent import MenuEvent
from premainEvent import PremainEvent
from mainEvent import MainEvent

class FactoryEvent:
    def eventConstructor(self, eventName):
        if eventName == 'Menu':
            return MenuEvent()
        elif eventName == 'preMain':
            return PremainEvent()
        elif eventName == 'Main':
            return MainEvent()
        else:
            raise ValueError(eventName)

```

FIGURE 8 – Factory pattern dans FactoryEvent

### 4.3 Composite

Le troisième design pattern que nous avons implémenté est une version du composite. Le but ici à été de concevoir une structure en arborescence : le `main_capi` utilise nos trois classes principales `menuEvent`, `premainEvent` et `mainEvent`. Ensuite, chacune de ces trois classes utiliseront à leur tour des classes pour gérer des sous-événements. L'idée, en organisant nos fichiers Event de cette manière, a été de bien structurer et de définir ce que chaque classe faisait.

On peut résumer notre arboressente comme suit :

- `main_capi` :
  - 1. `MenuEvent`
    - (a) `OptionEvent`
    - (b) `SetVolumeEvent`
  - 2. `PremainEvent`
    - (a) `SetNbPlayerEvent`
    - (b) `SetNameEvent`
    - (c) `SetBacklogEvent`
    - (d) `EraseBacklogEvent`
    - (e) `SetModeEvent`
    - (f) `SetChronoEvent`
  - 3. `MainEvent`
    - (a) `EndTaskEvent`
    - (b) `ExplicationEvent`
    - (c) `EndMainEvent`

## 5 Intégration continue

---

### 5.1 Tests Unitaires

Pour ce qui est des tests unitaires, nous avons opté par la solution proposée par notre encadrant de TD, "Pytest". Pytest est une librairie de tests très grandement inspirée syntaxiquement de JUnit. Étant donné que nous avons déjà rencontré, dans nos études, le langage Java, JUnit ainsi que des notions de tests unitaires plus avancés comme les mocks, pytest a été facile de prise en main.

Le but de cette classe a avant tout été de tester les méthodes relatives à la création, la recherche ainsi que l'enregistrement de modifications dans un fichier json. Nous aurions pu tester de plus complexes fonctionnalités, comme le clic ou le survol d'une case de notre application mais n'avons pas jugé cela important car déjà très visuel à l'utilisation de l'application.

En plus de cela, nous testons aussi l'unicité de notre singleton pattern ainsi que la création d'event avec notre factory pattern.

Certains tests sont dépendants d'autres tests unitaires positionnés en amont d'eux mêmes, nous avons donc implémenté une méthode de "skip" par décorateur fournie par pytest.

Vous pourrez retrouver toutes notre batterie de tests au sein du fichier "testUnitairtes.py" à la racine de notre projet. Ce fichier est entièrement commenté et il serait donc redondant de le re-mentionner avec des captures d'écran ici. Nous préférions donc le laisser en annexe dans notre code.

### 5.2 Génération de la documentation

Afin de mieux documenter notre projet (code), nous sommes passés par un workflow de documentation, Doxygen. Grâce à ce générateur de documentation, nous pouvons retrouver toutes les informations plus haut sur un site internet sur une branche "gh-pages" générée automatiquement à chaque push.

### 5.3 GitFlow

Avant de commencer notre projet, nous avons du choisir une manière de réaliser notre travail sans que l'on empiète sur le travail des autres. Pour cela, nous avons décidé de mettre en place une architecture type GitFlow. Nous avons séparé notre projet en 3 branches, main, develop et features.

La branche main devait recevoir toutes les versions stables du projet afin de proposer un rendu en fin de cycle de développement.

La branche develop devait accueillir toutes les fonctionnalités sur lesquels nous avons travaillé une fois ces dernières développées et fonctionnelles.

Enfin, les branches features, servaient à développer nos fonctionnalités dans des environnements sur lesquels on pouvait travailler sans venir empiéter sur le travail de l'autre. Il suffisait ensuite de "merge" nos branches à chaque itération afin de faire avancer le rendu final.

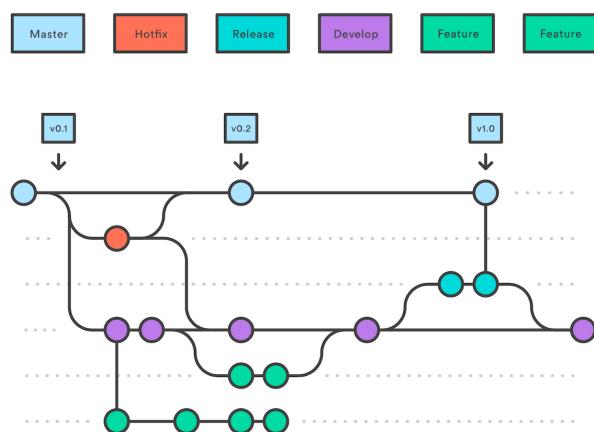


FIGURE 9 – Architecture GitFlow Grand projet