INP TOULOUSE ENSEEIHT

Projet long de TOB :
Rapport 3 *αnimal*

Younes Boudili, Romain Wiorowski, Pablo Neyens, Hossam El Markhi,
Thibaud Sarlat, Victor Mercier (1SN-K).

Département Sciences du Numérique - Première année

2019-2020

Table des matières

Table des figures

1 Introduction

αnimal est un jeu multijoueur qui simule les interactions au sein d'un écosystème animal. Le but de ce jeu est purement de divertir le joueur et de permettre à celui-ci d'adopter de nombreuses stratégies mais le jeu n'a pas pour but d'être pertinent d'un point de vue scientifique. Il est possible de jouer jusqu'à 12 joueurs, chaque joueur se verra donner une espèce animale parmi les 12 disponibles et essaiera de se débarrasser de tous ses adversaires pour devenir l'αnimal (vainqueur). Dans le cadre du sujet, le jeu doit être jouable sur un même ordinateur donc nous adopterons un système de tour par tour, les joueurs joueront sur le même interface les uns à la suite des autres. Pour préserver l'équilibre et une stabilité du jeu, les 12 espèces seront toujours mises en jeu et les espèces sans joueur seront gérées automatiquement.

2 Fonctionnalités principales

1. Observation globale du jeu à l'aide l'interface graphique et de la carte (itération 3)
2. Attaque des joueurs accessibles (dans la même zone) et déplacement du joueur sur la carte (itération 2)
3. Information sur l'état du joueur (itération 2)
4. Système d'expérience et arbre de compétences (itération 3)

3 Architecture

Le schéma suivant décrit d'une manière symbolique les différentes entités logicielles, et les liens structurels qui les mettent en relation.

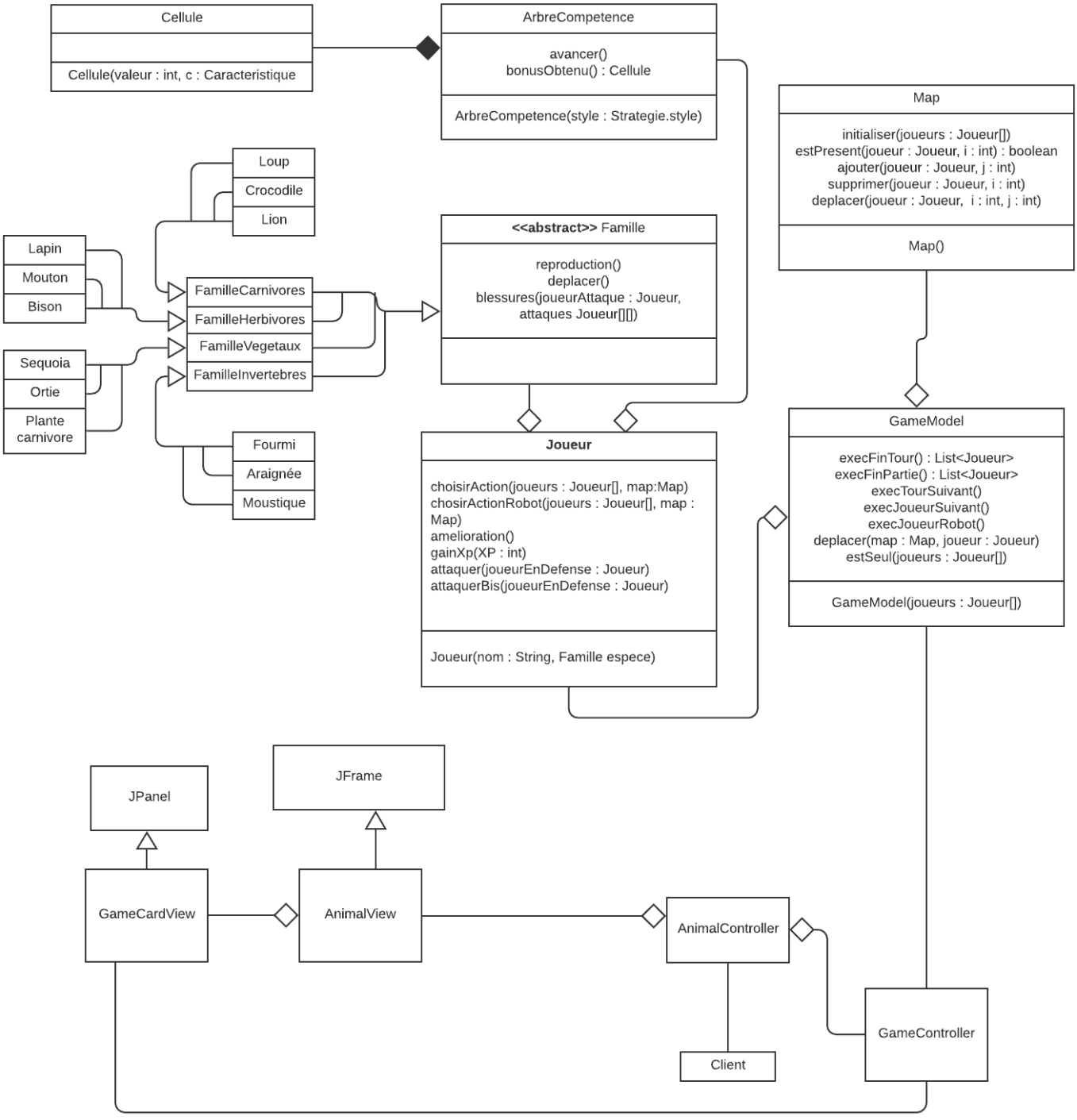

FIGURE 1 – Diagramme de classe de αnimal

On choisit de baser notre projet selon une architecture Modèle-Vue-Contrôleur. Dans un premier temps, le modèle et la vue ont été imaginées et développées indépendamment.
On détaille les rôles des différents classes principales associées à l'architectures MVC selon l'arborescence :

- **Client :** classe principale comportant la méthode main(). Elle lie AnimalModel et AnimalView au travers de AnimalController.
- **controllers/**
  - *AnimalController :* entrée des contrôleurs. C'est lui qui est instancié par Client, et qui va faire l'interface entre AnimalModel et AnimalView.
  - *IntroController :* contrôleur de la card Intro (on rappelle qu'on a découpé la vue en "cards", qui correspondent aux différentes pages de l'application, notamment la page d'introduction, et la page de partie). Il gère notamment la récupération et l'initialisation des joueurs (humain ou robot), et bascule sur la card Game en appelant GameController, et en affichant la card.
  - *GameController :* contrôleur de la card Game (la page du jeu). Il connaît GameView et GameModel. Il traite les changement d'états de la partie, et comporte un Listener sur les différents boutons de la vue. Les différents appels à GameModel permettent de modifier l'état du jeu.
- **view/**
  - *AnimalView :* entrée des vues. Il extends de JFrame, et constitue donc la fenêtre de l'application. Il comporte en attribut les différentes cards. Lors de son initialisation, il instancie ces différentes cards (en l'occurence 3 : defaultCardView celle par défaut qui est vide, introCardView et gameCardView). Une méthode setActiveCardView(CardView) permet d'afficher une card sur la fenêtre.
  - *CardView :* factorisation des cardviews. Ce sont bel et bien des vues, et extendent de JPanel. Ce sont elles qui sont chargées par AnimalView.
  - *cards/*
    - DefaultCardView : hérite de CardView et définit la vue et le JPanel associé à la card Default
    - IntroCardView : idem pour la card Intro
    - GameCardView : idem pour la card Game
- **model/**
  - *AnimalModel :* entrée des modèles
  - *GameModel :* modèle associé à la partie. C'est lui qui comporte l'état de la partie et les traitement directement associé, indépendamment de la vue et du contrôleur. Par exemple, qui contient la liste des joueurs, la carte, le joueur courant, le numéro du tour, le traitement des données lorsque le tour est terminé, ...
  - *IntroModel :* modèle associé à la card Intro.

4 Architecture du contrôleur

4.1 AnimalController

Dans la méthode main() de Client, on instancie le contrôleur principal avec animalView et animalModel. Elle instancie le contrôleur IntroController, affiche la vue associée.

4.2 IntroController

Il permet de récupérer les noms des joueurs entrés et instancie les Joueurs (en robot si nécessaire). Enfin, il crée un nouveau GameController qui gère l'affichage et le modèle de la partie jeu.

4.3 GameController

Il a pour attribut la vue de la card Game et a pour modèle gameModel. Il fait donc le lien entre ces deux éléments. Il gère aussi l'évolution des tours et des joueurs, ce qui est obligé car celui-ci est aussi lié aux événements liés aux boutons. Il transmet aussi à la vue les différentes modifications de l'état, notamment aux travers des méthodes update...(). Par exemple, lorsque l'on change de joueur, sont mises à jours toutes les données présentes dans la fenêtre. De même, lorsque l'arbre de compétences est modifié, les valeurs associées à l'espèce sont changées.

5 Architecture de la vue

5.1 AnimalView

C'est la vue englobant toutes les autres vues. C'est aussi la JFrame de l'application, et a comme attributs les différentes cards. Elles sont incluses dans un JPanel container. Des procédures d'initialisation permettent de créer les cards et de les ajouter au container. Une procédure setActiveCardView permet de sélectionner la card active.

5.2 CardView

(DefaultCardView, GameCardView et IntroCardView) Ce sont les vues correspondant aux différentes pages de l'application. Ce sont des JPanel. On y définit les différents composant du JPanel, et si besoin on ajoute les ActionListener.

5.3 GameCardView

La vue associée à la card Game. Celle-ci est décomposée en 4 sous JPanel, qui sont situés dans le package src.view.panels. On sépare aussi les onglets du JPanel en haut à droite.

6 Architecture du modèle

6.1 GameModel

La classe GameModel est la classe qui gère l'état global de la partie. C'est la classe qui fait passer les tours , qui fait jouer les joueurs, et qui termine la partie. Cette classe a aussi le contrôle sur la carte et peut déplacer les joueurs sur la carte.
Elle contient l'état du jeu. Les modifications sont appelées par GameController, et a notamment pour attributs la carte, les joueurs, le joueur en cours... On y trouve toutes les procédures qui permettent de gérer une partie à un tour et un joueur donné. On laisse le contrôleur se charger de l'évolution des tours et des joueurs, puisque ceux-ci sont aussi influencés par les événements fournis par la vue.

6.2 Famille

La classe abstraite Famille sert à gérer les différentes familles d'espèces. Elle définit les principales méthodes des familles, qui permettent de modifier les attributs d'une famille/espèce. Elle est spécialisée en **FamilleCarnivores**, **FamilleHerbivores**, **FamilleInvertebres**, et **FamilleVegetaux** qui définissent chacune les statistiques propres à leur famille.Ces statistiques sont les attributs de population, nourriture, hostilité, résilience, etc.
La classe contient également une méthode reproduction() qui permet de mettre à jour le niveau de la population à chaque tour.

6.3 Les familles

Les différentes familles, héritant de la classe Famille sont : *FamilleCarnivores*, *FamilleHerbivores*, *FamilleInvertebres*, et *FamilleVegetaux*.
Elles définissent les statistiques propres à leur famille, comme leur alimentation.
L'alimentation d'une famille est son efficacité pour attaquer les autres familles. Elle est définie comme suit (l'efficacité d'attaquer une espèce de la même famille est de 5 %) :

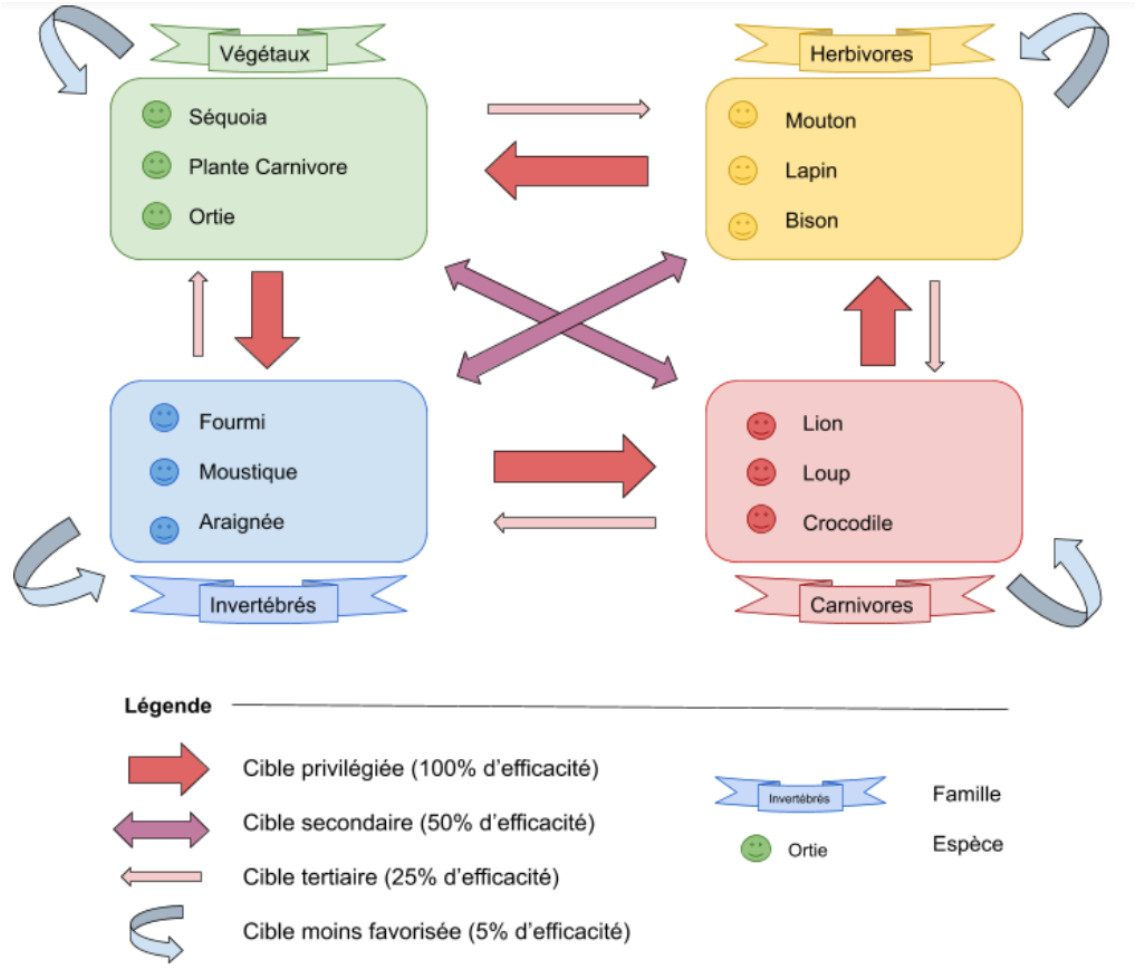

FIGURE 2 – Écosystème de αnimal

Chaque famille possède une capacité spéciale, dont héritent toutes les espèces appartenant à celle-ci :

- **Végétaux :** Gagnent 1 de nourriture chaque jour sans attaquer. Invertébrés : Gagnent un bonus d'hostilité par rapport à la population (bonus de 0.5% de leur population).
- **Herbivores :** Gagnent un bonus de reproduction par rapport à la population (bonus de 10% de leur population).
- **Carnivores :** Quand ils exterminent (attaquer une espèce dont la population tombe à 0 lors du tour) une espèce ils ont un bonus sur l'un de leur attributs (cet attribut est re-défini pour chacune des espèces qui héritent de cette famille) .

6.4 Les espèces

Les classes espèces (Araignee, Bison, etc.) définissent les différentes espèces qu'il est possible de jouer. Elle héritent des classes *FamilleCarnivores*, *FamilleHerbivores*, *FamilleInvertebres*, et *FamilleVegetaux*, et redéfinissent leurs attributs, en fonction de leurs compétences. Les attributs sont définis comme suit :

ATTRIBUTS	Nom	Population	Résilience	Reproduction	Hostilité	Déplacement
ESPÈCES						
VÉGÉTAUX						
	Séquoia	12	8	2	3	6
	Plante Carnivore	40	2	4	8	6
	Ortie	100	4	8	4	5
INVERTEBRES						
	Fourmi	300	0,4	20	5	4
	Moustique	350	0,1	45	5	3
	Araignée	150	0,3	8	8	4
HERBIVORES						
	Mouton	100	0,5	20	5	5
	Lapin	40	0,3	50	2	1
	Bison	20	5	4	8	5
CARNIVORES						
	Lion	20	2	6	14	3
	Loup	30	1	7	10	3
	Crocodile	10	5	4	14	4

FIGURE 3 – Table des attributs

Avec :

-
- **Population :** Nombre d'individus en vie de l'espèce ;
- **Résilience :** Dégâts nécessaires pour tuer un individu de l'espèce ;
- **Reproduction :** Nombre de nouveaux individus à chaque tour ;
- **Hostilité :** Dégâts infligés lors d'une attaque ;
- **Déplacement :** Nombre de tours nécessaires pour un déplacement.

On a, de plus, les deux attributs suivants :

- **Défense** (initialisée à 0 pour toutes les espèces) : Est soustraite à l'hostilité d'une autre espèce, pour diminuer son impact ;
- **Nourriture** (initialisée à 9 pour toutes les espèces) : Permet aux espèces de se reproduire à chaque tour, selon différents paliers. Le niveau de population augmente à chaque tour selon les paliers de nourriture :
  - *supérieure à 10 :* + 1.5 * reproduction,
  - *entre 5 (exclus) et 10 :* + 1 * reproduction,
  - *entre 3 (exclus) et 5 :* + 0.5 * reproduction,
  - *entre 1 (exclus) et 3 :* + 0,
  - *inférieure ou égale à 1 :* - 0.5 * reproduction.

En plus des capacités spéciales des familles, chaque espèce possède aussi sa propre particularité :

- **Séquoia :** Gagne 1 de défense tous les 5 tours.
- **Plante Carnivore :** Augmente l'efficacité de 5% de l'espèce mangée, de manière permanente.
- **Ortie :** Lorsqu'une espèce attaque l'ortie, elle perdra de la population proportionnellement à la population des orties (population des orties /(100*résilience)).
- **Fourmi :** Lorsque la population des fourmis tombe à zéro pour la première fois, elles renaissent avec une nouvelle reine, leur population est mise à 100.
- **Moustique :** Lorsque les moustiques attaquent un joueur, ce joueur est empoisonné pendant un tour, une cible empoisonnée verra échouer son attaque avec une probabilité de 1/3.
- **Araignée :** Lorsque les araignées attaquent un joueur, ce joueur est immobilisé, il ne pourra pas enclencher de déplacement lors de son prochain tour (mais ses déplacements en cours sont préservés).
- **Mouton :** Gagne un bonus de défense par rapport à la population (bonus de 1%).
- **Lapin :** Lorsque le lapin se reproduit , il accède automatiquement au palier de reproduction strictement supérieur à celui permis par sa nourriture. Si le lapin a déjà accès au palier maximal alors il ne se passe rien.
- **Bison :** Lorsque une espèce d'une famille **x** attaque le bison, celui-ci gagne une efficacité contre la famille **x** de 150% lors du tour suivant pour pouvoir se venger.

Pour les espèce qui héritent de la famille des carnivores, il gagnent un bonus lorsqu'ils exterminent une espèce :

- **Lion :** +5 en hostilité.
- **Loup :** +5 en reproduction.
- **Crocodile :** +3 en défense.

6.5 Joueur

La classe Joueur sert à gérer l'état global du Joueur. Un joueur est attribué d'un nom de son espèce, de son style de jeu, de son arbre de compétence, de son expérience et de ses coordonnées sur la map. Il est aussi attribué d'un booléen qui permet de savoir si le joueur est un robot ou non. La classe définit plusieurs méthodes d'action que le joueur peut réaliser au cours de la partie comme *attaquer()* ou *reproduction()*. Cette classe gère aussi le gain d'expérience pour le joueur et les améliorations liées à l'arbre de compétence.

6.6 ArbreCompetence

C'est la classe qui gère l'arbre de compétence du joueur. Elle définit ainsi les améliorations qu'un joueur peut posséder et lesquelles il peut débloquer.

6.7 Map

La carte du jeu est composée de 4 zones sous forme de listes.

```
private ArrayList<Joueur> zone1;
private ArrayList<Joueur> zone2;
private ArrayList<Joueur> zone3;
private ArrayList<Joueur> zone4;
```

Toutes les familles en jeu se trouveront dans une des 4 zones. La majorité des interactions entres familles (comme les combats) se feront entres familles d'une même zone. En manipulant les zones de la carte, on utilise des entiers. Ce qui explique la méthode :

```
public ArrayList<Joueur> intToZone(int i)
```

6.8 Autres

6.8.1 Package lib

Il contient des ré-écriture de classes utiles, notamment pour la vue.

6.8.2 Ressources graphiques

Il contient les ressources utiles à la vue.

7 Difficultés

On eu a des difficultés lors de la mise en place du système d'expérience pour équilibrer le jeu. Pour mettre en place les différentes capacités des espèces et des familles, qui peuvent agir sur des aspects très différents du jeu, nous avons rencontré quelques difficultés. Pour les surmonter, nous avons mis en place des méthodes redéfinies par chaque espèce, qui sont appelées à chaque tour.
La gestion et l'équilibrage des différentes espèces vis à vis des énormes écarts de population fut un problème, nous avons mis en place un facteur résilience propre à chaque espèce pour rétablir un équilibre.

8 Organisation de l'équipe et mise en oeuvre des méthodes agiles

- Répartition des tâches
- Réunions régulières sur l'avancée du projet
- Définition de buts intermédiaires (calendrier, objectifs)
- Outils pour l'organisation et la communication (discord, trello...)
- Discussion/ explication pour résoudre les problèmes et les incompréhensions entre les membres de l'équipe

Afin d'être efficaces dans le travail à réaliser, nous nous sommes organisés en nous répartissant les tâches entre nous, tout en communiquant régulièrement sur notre avancée, afin de rester informés de l'état du projet.
Nous avons aussi organisé des réunions régulières, avec l'ensemble de l'équipe, pour faire un point sur notre progression, et ainsi pouvoir se fixer des nouveaux objectifs, clairs et réalisables.
Ces discussions étaient l'occasion d'expliquer les choses réalisées par chacun, et ainsi résoudre les problèmes et les incompréhensions entre les membres de l'équipe.
Ces différents objectifs étaient planifiés dans un calendrier général, pour avoir une gestion des différentes étapes du projet dans le temps, et ainsi pouvoir réaliser les objectifs généraux initiaux.
Comme nous avons beaucoup travaillé en équipe, nous utilisions des outils de communication et d'organisation, comme discord ou trello.
Dans le but d'éviter les conflits de version du code entre nos différentes modifications, nous informions le reste de l'équipe à chaque modification envoyée.