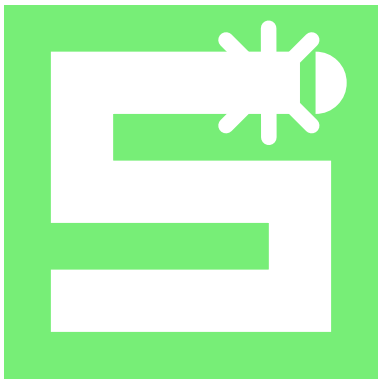

Rapport de gestion du projet Stibbons

5 juin 2015

Julia Bassoumi - julia.bassoumi@etud.univ-montp2.fr
Florian Galinier - florian.galinier@etud.univ-montp2.fr
Adrien Plazas - adrien.plazas@etud.univ-montp2.fr
Clément Simon - clement.simon@etud.univ-montp2.fr

Encadrant : Michel Meynard

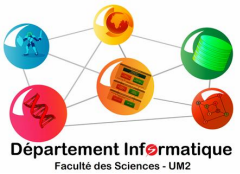

Table des matières

1 Sujet **2**

2 Backlog initial **4**

3 Backlog final **6**

4 Données quantitatives **8**
4.1 Git 8
4.2 Réalisation 9

5 Conclusion **11**

Chapitre 1

Sujet

Le projet Stibbons a pour but la création d'un environnement de programmation multi-agents pour développeurs de tout niveau. NetLogo (cf. Figure 1.1) a été un modèle lors de la conception de Stibbons, et une des contraintes alors fixées était d'avoir une application similaire mais exécutant les agents de façon parallèle et non séquencielle comme le fait NetLogo.

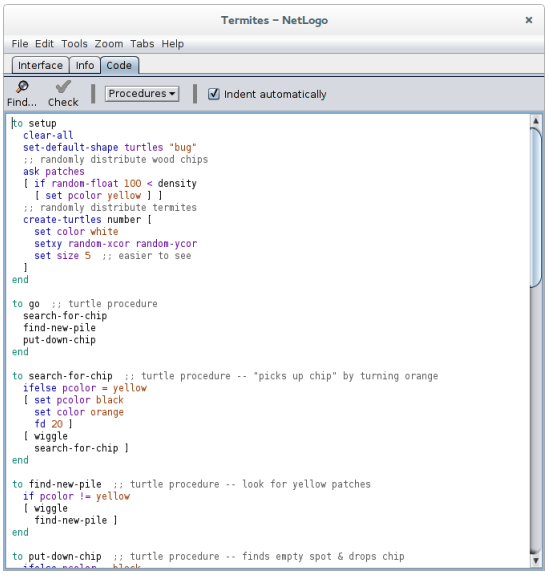

FIGURE 1.1 – L'éditeur de texte de NetLogo

La finalité d'un tel environnement est de permettre d'effectuer des simulations de comportements d'agents. Un exemple classique est celui dit « des termites » (cf. Figure 1.2), qui consiste à modéliser des termites ramassant des brindilles pour former leur termitière. Pour cela, l'utilisateur définit le comportement d'agents représentant ici les termites, étale des brindilles sur le sol, et observe l'agissement des agents faire évoluer le modèle.

Nous avons décidé de créer un langage permettant de manipuler des agents mobiles (dits tortues) afin de les faire se déplacer et communiquer entre eux, soit directement par envoi de messages, soit indirectement en modifiant et analysant leur environnement. Une interface graphique permettant d'observer directement l'évolution du modèle était également prévue.

Au cours du projet, notre encadrant, Michel Meynard, nous a suggéré de permettre l'export du modèle en cours d'exécution (comprenant l'état du monde, des zones le constituant, et des tortues y évoluant), fonctionnalité que nous avons donc rajouté à notre backlog, et à terme à notre application.

Plus tard, il a aussi été décidé de proposer un programme complémentaire à notre application, qui serait utilisable en ligne de commande et ne nécessiterait pas l'utilisation d'un serveur

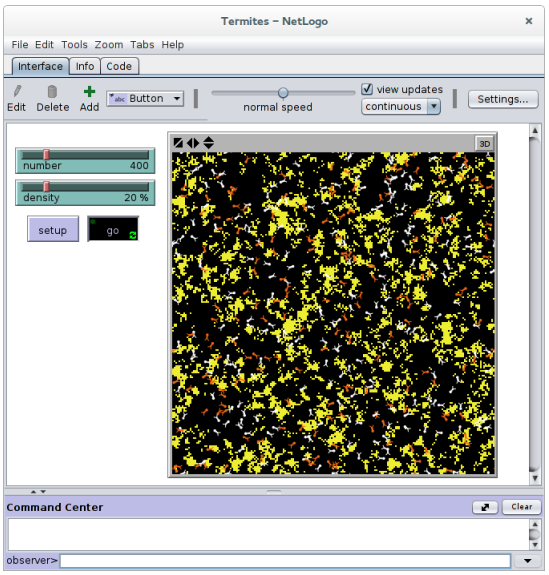

FIGURE 1.2 – La simulation des termites dans NetLogo

graphique pour fonctionner, mais permettant d'exporter le modèle à intervalle régulier. Cela permet de ne pas utiliser les ressources graphiques de l'ordinateur, d'exécuter la simulation à pleine vitesse et de l'effectuer sur une ferme de calcul.

Nous avons également décidé d'intégrer un éditeur de texte à l'application principale afin de pouvoir directement éditer et exécuter un programme Stibbons.

Chapitre 2

Backlog initial

Nous avons très rapidement choisi de réaliser notre TER avec la méthode agile SCRUM, car en plus d'être aujourd'hui de plus en plus utilisée en entreprise, elle nous permettait d'avoir une version de notre projet fonctionnelle à la fin de chaque sprint. La durée des sprints a été fixée à deux semaines, car cela nous semblait être un bon compromis entre le temps nécessaire à l'ajout de fonctionnalités conséquentes et des itérations rapides.

Lors du sprint 0, nous avons effectué une analyse des outils à notre disposition (Flex, Bison, etc.) et de projets similaires existants tels que le langage Logo et ses dérivés. Nous avons ensuite réalisé une estimation du temps de travail (cf. Figure 2.1) afin d'évaluer combien de sprints nous pourrions faire en tenant compte d'autres activités (cours, examens, révisions, etc.).

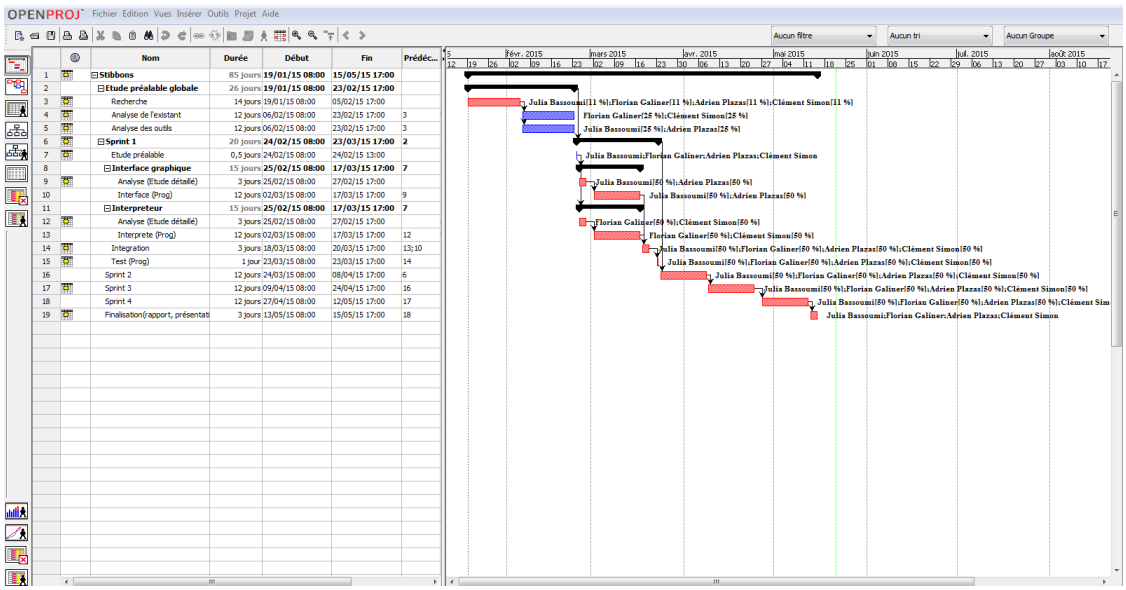

FIGURE 2.1 – Planning OpenProject

Après réunion avec notre encadrant, nous avons également défini les différentes fonctionnalités (sous forme de scénarios utilisateurs) que nous souhaitions voir dans notre future application, associant à chacune d'entre elles une priorité, et avons établi à partir de là le backlog initial du projet (cf. Table 2.1).

Le projet a été développé semblablement à un projet interne d'entreprise, En effet notre encadrant ne jouait pas le rôle d'un client mais plutôt celui d'un patron.

Nous effectuions une réunion en début de chaque sprint, au cours de laquelle nous commencions par choisir les tâches devant être réalisées durant le sprint selon leurs priorités, mais également selon d'autres facteurs tels que l'ordre de réalisation ou le temps de travail nécessaires.

L'estimation du volume horaire nécessaire à chaque sprint était réalisée à l'aide de petits

papiers sur lesquels chacun d'entre nous mettait ses estimations minimale et maximale, avant de les mettre en commun pour en discuter. L'estimation finale correspondait à la moyenne entre la plus haute et la plus petite de toutes les estimations.

Lors du premier sprint, nous nous sommes réparti le travail selon les envies de chacun ; il est cependant vite apparu que deux groupes de deux personnes ressortaient : un groupe centré sur le modèle et l'interface et un groupe centré sur la grammaire et l'interprète. À chaque sprint, chacun avait une ou plusieurs tâches à effectuer, parfois seul, parfois en collaboration.

id	Scénario utilisateur	Priorité	Tests	Estimation	Sprint	Statut	Temps réel
1	L'utilisateur écrit du code dans un éditeur	200					
2	L'utilisateur importe du code dans le logiciel	1100	Importer du code Stibbons depuis un fichier, vérifier que le code obtenu est bien identique à celui du fichier.	4h	1	Fini	1h
3	L'utilisateur visualise les rapports d'erreurs du code	400					
4	L'utilisateur visualise l'évolution du modèle	1400	Vérifier que l'interprétation d'instructions données fait bien évoluer comme prévu la tortue dans son environnement.	24h	1	Fini	70h
5	L'utilisateur modifie la vitesse (pause, pas à pas, parallèle)	700					
6	Le « dieu-tortue » interprète le code de l'utilisateur	1500	Lancer l'interprétation pour : repeat 4 fd 1 rt 90 (suivant syntaxe) ainsi que pour du code avec des erreurs : repat 4 ... par exemple ou repeat 4 fd 1 rt	12h	1	Fini	32h
7	L'utilisateur crée une nouvelle tortue (avec un code)	600	Lancer l'interprétation pour : create-turtle et observer une nouvelle tortue apparaître dans l'interface graphique.	7,5h	2		
8	Les tortues s'exécutent en parallèles	1200	Lancer l'interprétation pour deux tortues d'un bout de code et observer l'exécution parallèle via des écritures dans le terminal (Je suis tortue 1 et Je suis tortue 2 par exemple)	42,5h	2		
9	Les tortues communiquent directement entre elles	900					
10	Une tortue se déplace dans l'environnement	1300	Ecriture d'instructions simples : repeat 4 fd 1 rt 90 (suivant syntaxe) - Renvoi de la position de la tortue après chaque deplacement : where_am_i() ; (suivant syntaxe)	16h	1	Fini	24h
11	Les tortues communiquent avec les zones de l'environnement	1000					
12	L'utilisateur exporte le code	500					
13	L'utilisateur exporte le modèle	300					
14	L'utilisateur ajoute une entrée	100					
15	L'utilisateur remet à zéro l'environnement	800					
16	L'utilisateur utilise des variables dans le code	1275	Ecrire a = 90 fd a et observer la tortue qui avance.	12h	2		
17	L'utilisateur définit des fonctions personnalisées dans le code	1250	Ecrire function f () fd 90 f () et observer la tortue qui avance.	23,5h	2		
18	Les tortues communiquent via l'environnement	950					
19	L'utilisateur utilise des conditionnelles	550	Ecrire if(false) fd 90 et observer que la tortue ne bouge pas. Refaire le même test avec if(true) et observer que la tortue bouge.	3,5h	2		
20	L'utilisateur utilise des boucles	575	Ecrire pd repeat 4 fd 40 rt 90 et observer que la tortue dessine un carré.	7h	2		

Table 2.1: Le backlog à la fin du sprint 1

Chapitre 3

Backlog final

On peut voir sur le backlog final (cf. Table 3.1) l'avancée du projet après le sprint 5.

Tout au long de la réalisation du projet, de nouvelles idées de fonctionnalités sont apparues, que nous avons rajoutées au backlog sous forme de tâches à effectuer et de scénarios utilisateurs.

L'estimation du temps nécessaire était particulièrement compliquée lors des premiers sprints, ce qui peut être noté en comparant les écarts entre les estimations de temps nécessaire et le temps réel passé sur chaque tâche. Nous étions globalement dans les temps à chaque sprint, en effet très peu de tâches ont été finies en retard et reportées au sprint suivant.

id	Scénario utilisateur	Priorité	Tests	Estimation	Sprint	Statut	Temps réel
1	L'utilisateur écrit du code dans un éditeur	200	L'utilisateur écrit du code dans un éditeur intégré	20h	5	Fini	19h
2	L'utilisateur importe du code dans le logiciel	1100	Importer du code Stibbons depuis un fichier, vérifier que le code obtenu est bien identique à celui du fichier.	4h	1	Fini	1h
3	L'utilisateur visualise les rapports d'erreurs du code	400	Exécuter pd 50 et constater une erreur.	8h	4	Fini	8h
4	L'utilisateur visualise l'évolution du modèle	1400	Vérifier que l'interprétation d'instructions données fait bien évoluer comme prévu la tortue dans son environnement.	24h	1	Fini	70h
5	L'utilisateur modifie la vitesse (pause, pas à pas, parallèle)	700	Faire varier la vitesse et observer le changement dans l'execution des tortues.	12h	4	Fini	14h
6	Le « dieu-tortue » interprète le code de l'utilisateur	1500	Lancer l'interprétation pour : repeat 4 fd 1 rt 90 (suivant syntaxe) ainsi que pour du code avec des erreurs : repat 4 ... par exemple, ou repeat 4 fd 1 rt	12h	1	Fini	32h
7	L'utilisateur crée une nouvelle tortue (avec un code)	600	Lancer l'interprétation pour : create-turtle et observer une nouvelle tortue apparaître dans l'interface graphique.	7,5h	2	Fini	4h
8	Les tortues s'exécutent en parallèles	1200	Lancer l'interprétation pour deux tortues d'un bout de code et observer l'exécution parallèle via des écritures dans le terminal (Je suis tortue 1 et Je suis tortue 2 par exemple)	42,5h	2	Fini	40h
9	Les tortues communiquent directement entre elles	900	Ecrire send(t,"Je suis là") avec t une tortue qui a pour code : m = recv() if (m == "Je suis là") fd 50 et observer la tortue avancer	30h	3	Fini	16h
10	Une tortue se déplace dans l'environnement	1300	Ecriture d'instructions simples : repeat 4 fd 1 rt 90 (suivant syntaxe) - Renvoi de la position de la tortue après chaque deplacement : where_am_i() ; (suivant syntaxe)	16h	1	Fini	24h
11	Les tortues communiquent avec les zones de l'environnement	1000	Ecrire if(zone.color == red) color = blue sur une zone de couleur rouge et observer la tortue changer de couleur.	30h	3	Fini	12h
12	L'utilisateur exporte le code	500	L'utilisateur sauvegarde son code dans un fichier externe	1h	5	Fini	1h
13	L'utilisateur exporte le modèle	300	Sauvegarder le modèle et observer les données en sortie (JSON)	30h	4	Fini	22h
14	L'utilisateur ajoute une entrée	100					
15	L'utilisateur remet à zéro l'environnement	800	Remettre à zéro après une exécution et observer que le monde est vierge.	20h	4	Fini	23h

Table 3.1: Le backlog à la fin du sprint 5

id	Scénario utilisateur	Priorité	Tests	Estimation	Sprint	Statut	Temps réel
16	L'utilisateur utilise des variables dans le code	1275	Ecrire a = 90 fd a et observer la tortue qui avance.	12h	2	Fini	3h
17	L'utilisateur définit des fonctions personnalisées dans le code	1250	Ecrire function f () fd 90 f () et observer la tortue qui avance.	23,5h	2	Fini	11h
18	Les tortues communiquent via l'environnement	950	Ecrire broadcast(20,"Je suis là") et dans une autre tortue dans le rayon avec le code m = recv() if (m == "Je suis là") fd 50 et observer la tortue avancer.	10h	3	Fini	2h
19	L'utilisateur utilise des conditionnelles	550	Ecrire if(false) fd 90 et observer que la tortue ne bouge pas. Refaire le même test avec if(true) et observer que la tortue bouge.	3,5h	2	Fini	3,5h
20	L'utilisateur utilise des boucles	575	Ecrire pd repeat 4 fd 40 rt 90 et observer que la tortue dessine un carré.	7h	2	Fini	3,5h
21	L'utilisateur définit des fonctions avec paramètres	1225	Ecrire function f(a) fd a f(90) et observer la tortue avancer.	10h	3	Fini	16h
22	L'utilisateur instancie des agents avec paramètres	560	Ecrire agent wolf (a) fd a new wolf (50) et observer la nouvelle tortue avancer.	10h	3	Fini	16h
23	L'utilisateur modifie la couleur d'un agent (tortue ou zone)	450	Ecrire new agent color = red et observer la nouvelle tortue rouge.	3h	3	Fini	2h
24	La tortue accède aux données de son parent	50	Taper parent.color= blue et observer le parent devenir bleu	1h	5	Fini	1h
25	L'utilisateur peut stocker des valeurs dans une table	540	On stocke le retour dans t= 1,2	10h	4	Fini	9h
26	L'utilisateur parcours un tableau	530	Faire foreach(f : 1,2) println(f)	4h	5	Fini	4h
27	L'utilisateur lance l'application sans interface graphique	250	Lancer l'application et regarder les fichiers d'exportations	16h	5	Fini	15h

TABLE 3.1: Le backlog à la fin du sprint 5

Chapitre 4

Données quantitatives

4.1 Git

Afin de permettre une meilleure gestion du projet (travail parallèle, gestion de bugs, etc.) nous avons décidé d'utiliser le gestionnaire de version Git ainsi que la forge GitLab mise à la disposition des étudiants par le SIF. La prise en main fut facile, les membres du groupe ayant pratiquement tous déjà utilisé cet outil. Nous avons également utilisé le système de suivi de bugs intégré à Gitlab (cf. Figure 4.1).

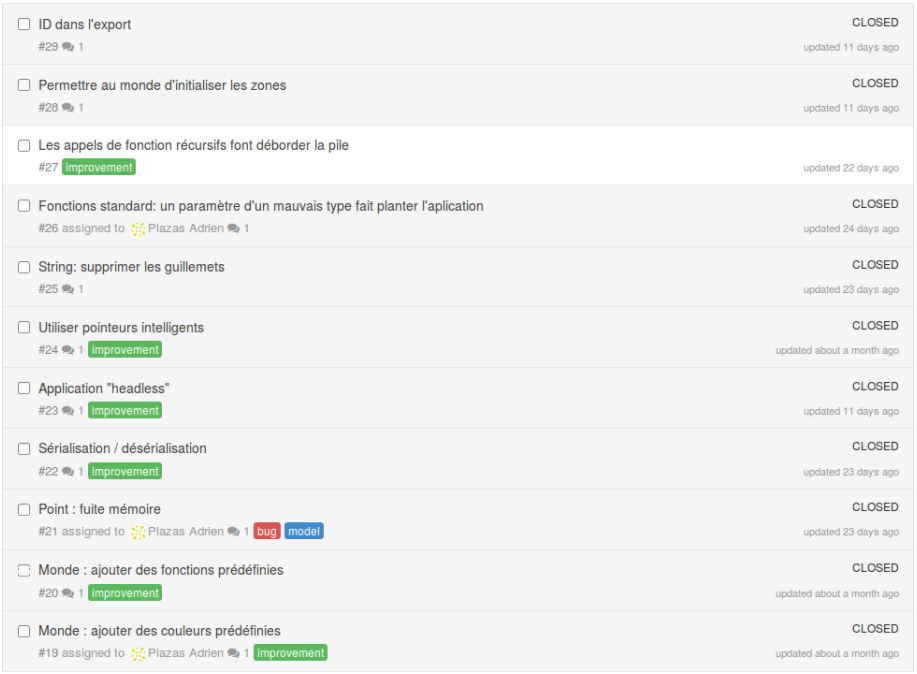

FIGURE 4.1 – Système de suivi de bugs de Gitlab

Notre organisation des branches fut la suivante :

— la branche master devait contenir une version fontionnelle compilable ;
— des branches de développement étaient créées pour chaque nouvelle fonctionnalité, et n'étaient fusionnées sur la branche master que lorsqu'elles étaient pleinement fonctionnelles ;
— des branches spécifiques à chaque version ont été dérivées de master lors de l'officialisation desdites versions.

Nous utilisions également Gitg, pour avoir une meilleure vue de l'état de notre dépôt (cf. Figure 4.2).

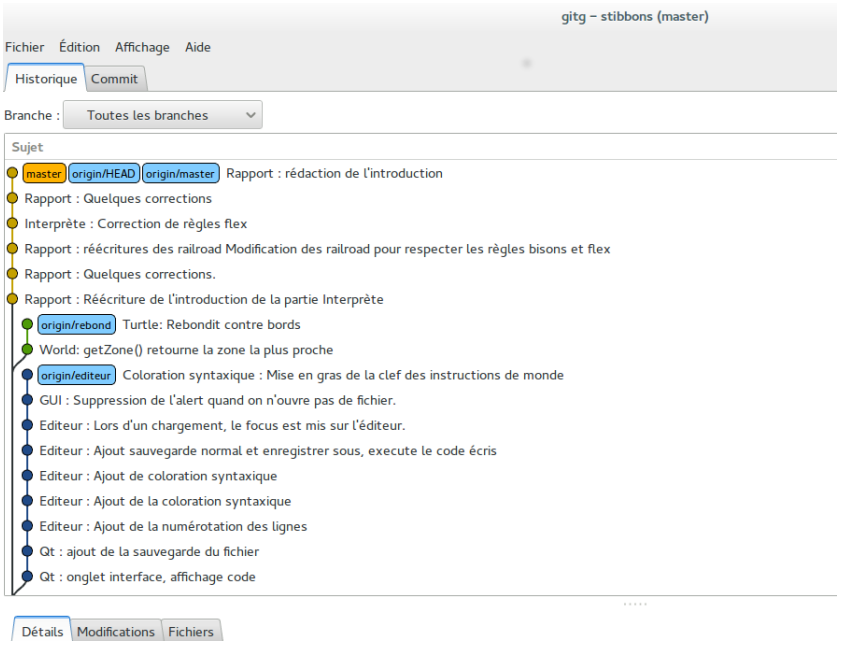

FIGURE 4.2 – Vue des branches dans Gitg

Lors de la sortie de la version 1.0, le dépôt comprenait 493 commits, avec une moyenne de 4,7 commits par jour, soit 98,6 commits par version en moyenne, 12 branches et aucune étiquette (nous avons utilisé les branches afin d'étiqueter les versions). Il y avait 11 827 lignes de code réparties en 324 lignes pour l'application console, 1 496 lignes pour l'application graphique, 3 027 lignes pour l'interprète, 6 188 lignes pour le modèle et 792 lignes pour les tests.

4.2 Réalisation

Le backlog final nous permet de constater que nous avons passé 372h à développer l'application, soit 93h par personne et environ 13,5 j/homme. Nous pouvons ainsi estimer un total d'environ 400h pour 4 pour le projet entier (réunions comprises), soit 14,5 j/homme. L'UE TER M1 prévoyant 50h par personne (soit 200h pour 4), on peut noter que ce quota a été doublé.

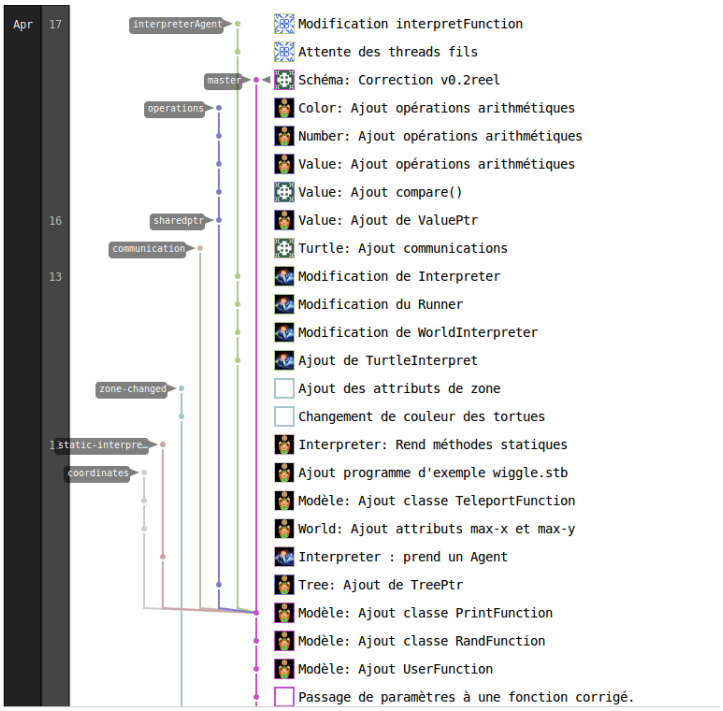

FIGURE 4.3 – Vue des branches lors de la version 0.3 sur Gitlab

Chapitre 5

Conclusion

Au cours de ces quatre mois, nous avons beaucoup appris sur SCRUM et l'Agile, grâce au projet mais également grâce à Sandrine Maton, intervenante extérieure à l'UM.

Le planning a été respecté, avec des sprints réguliers de deux semaines tout en faisant évoluer nos objectifs. Au cours de cette évolution, nous avons senti une réelle cohésion de groupe se former, notamment à partir du troisième sprint où le fait de travailler avec une méthode agile permis de mieux s'organiser et nous incita fortement à communiquer.

L'ajout de tâches au backlog n'a pas perturbé notre rythme de développement et nous avons fini notre projet dans les temps, avec quelques fonctionnalités supplémentaires initialement non prévues.

De plus nous avons des idées d'améliorations à proposer pour une potentielle version 2.0.