

ISIMA DEUXIÈME ANNÉE

FILIÈRE GÉNIE LOGICIEL

Projet de simulation Multi-Agents

Pierre CHEVALIER
Pierre-Loup PISSAVY

Enseignant :
David HILL

2015 – 2016



Table des matières

1	Présentation	2
2	Développement	5
2.1	Patrons de conception	6
2.1.1	Patron composite	6
2.1.2	Patron stratégie	7
2.1.3	Singleton	8
2.2	Présentation des interfaces	8
2.2.1	Interface console	8
2.2.2	Interface Graphique	10
3	Conclusion	11
A	Utilisation de Git	13
A.1	Résumé	13
A.2	Procédure d'installation	14
A.3	Présentation des fonctionnalités	15
A.3.1	Git dans le terminal	15
A.3.2	Git avec Github Desktop	16
A.3.3	Git avec Visual Studio	17
B	Doxygen – Présentation d'un outil de documentation	19
B.1	Présentation	19
B.2	Installation	19
B.2.1	Logiciel	19
B.2.2	Plug-ins	20
B.3	Fonctionnalités proposées	20
B.3.1	Réglage du rendu	20
B.3.2	Réglage de l'extraction de documentation	21
B.3.3	Et dans le code...	21
B.4	Utilisation au sein du TP Multi-agents	22
B.5	En bref	23
C	Résultats	24
C.1	Résultats de cent simulations avec deux factions	24
C.1.1	Moyenne faction gagnante	24
C.1.2	Moyenne autres factions (éliminées)	25
C.2	Interprétation	25

1 | Présentation

Ce TP de Simulation s'inscrit également dans le cadre du cours d'Architectures Logicielles et Qualité. Il s'agit de concevoir un programme de simulation multi-agents. Le sujet étant libre, nous avons choisi d'imaginer un monde composé de planètes. Ces planètes appartiennent à des factions qui s'affrontent mutuellement.

Le monde est par défaut une grille carrée de 400 cases (20×20). On peut travailler avec une grille torique (en employant la méthode d'initialisation des voisins `void Virtual_planet::set_neighbourhood2()` en lieu et place de `void Virtual_planet::set_neighbourhood()`).

Le monde propose une liste d'agents en attente, et une liste des agents ayant déjà joué. Afin de définir un ordre aléatoire dans les attaques ou actions des agents, la liste des agents en attente est mélangée aléatoirement. Entre deux tours, la liste des agents ayant déjà joué est utilisée pour déterminer la liste des agents en attente du prochain tour.

Chaque planète peut choisir de mener une attaque afin d'étendre l'espace colonisé par la faction à laquelle elle appartient, ou bien de produire des richesses qui permettront une expansion ultérieure (car celle-ci a un coût).

Le coût d'une attaque est estimé selon le taux de défense de l'adversaire.

Lorsqu'une planète subit une attaque, elle peut tenter de parer à celle-ci. Cette parade est fonction des capacités de défense de la faction. Si elle n'a pas les capacités suffisantes, alors elle subit, ce qui a pour effet de réduire ses capacités de défense et le rendement de ses productions (dégâts et pertes).

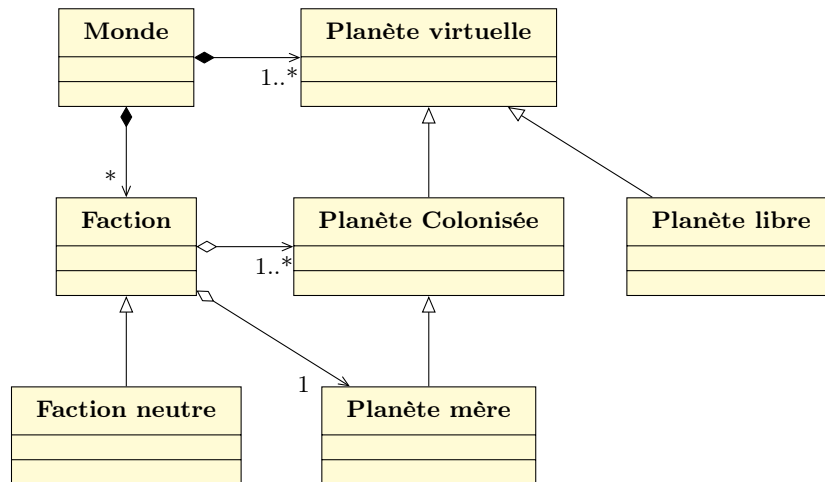
Lorsque la planète n'adopte pas de comportement offensif, elle choisit de rendre son industrie plus pérenne et améliore ses productions (c'est le cas lorsque l'attaque a échoué). Les productions sont définies en fonction de l'expertise de la colonie et du taux de production de la planète (certaines planètes sont plus propices à l'activité industrielle).

Lorsqu'une planète libre est colonisée, alors elle est remplacée par une planète colonisée appartenant à la faction de l'attaquant. Si toutefois cette planète appartenait à une faction adverse, cette faction la perd et la nouvelle faction peut alors en profiter.

Lorsqu'une planète est attaquée, et qu'elle n'a pas encore joué, son action est annulée, car elle serait alors susceptible de s'en prendre à sa propre faction, ce qui est interdit.

Certaines planètes ont un statut spécial : les planètes mères. Il y a une planète mère par faction, elle agit comme une colonie, mais provoque la perte totale et définitive de la faction si jamais elle ne parvient pas à se défendre lors d'une attaque.

Nous avons finalement retenu le diagramme de classes général suivant :



Pour réaliser ce projet, nous avons décidé d'avoir recours à plusieurs outils, et parfois d'utiliser différents outils pour parvenir à la même fin, ceci pour nous rendre compte des problèmes que nous pouvons rencontrer en entreprise lorsque plusieurs développeurs travaillent sur un même projet avec des outils différents. C'est pourquoi nous avons utilisé plusieurs IDE : Geany à l'ISIMA, QtCreator sur Windows et sur Linux, et Microsoft Visual Studio (2013 et 2015) sur Windows.

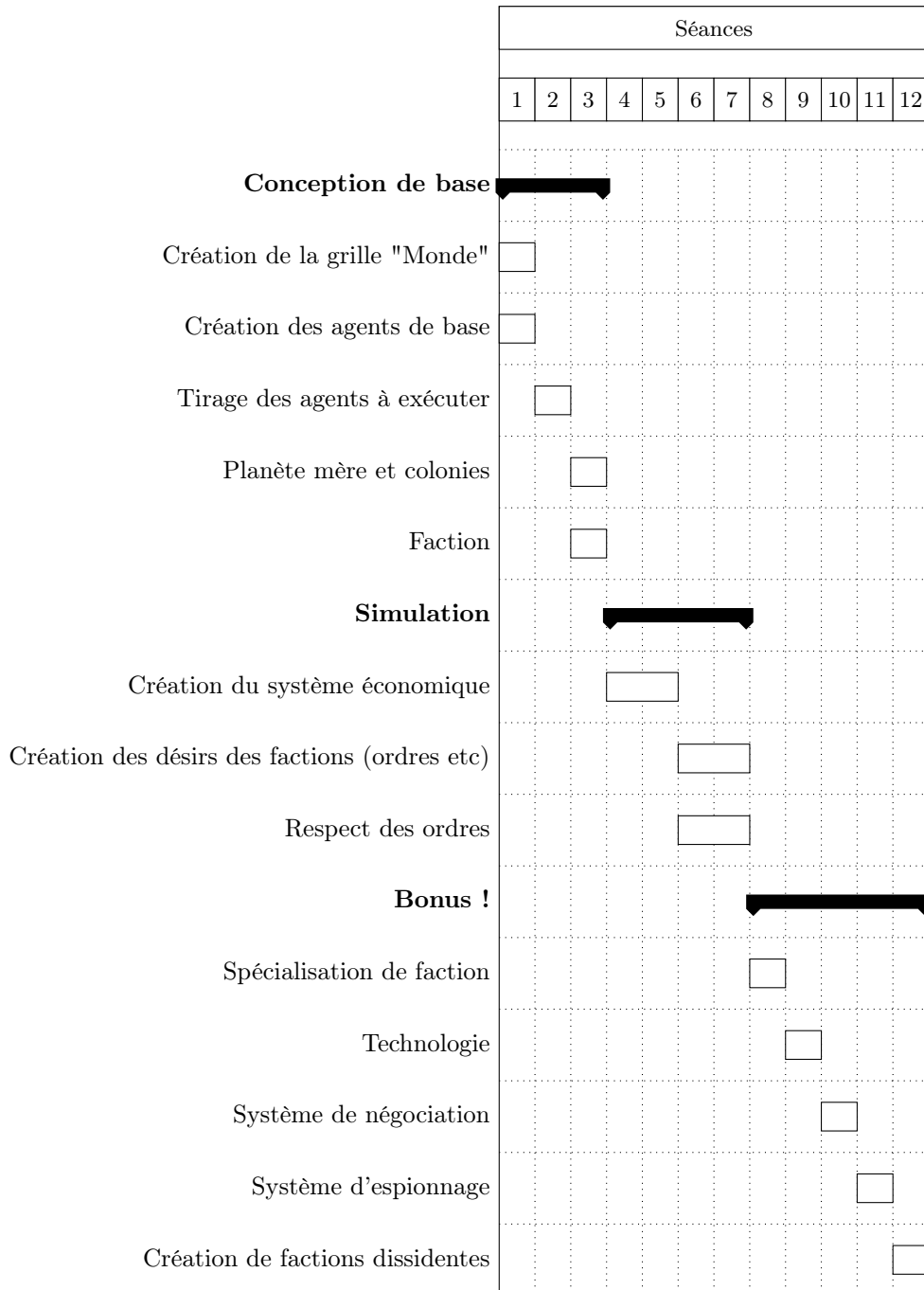
Chaque IDE propose différentes fonctionnalités plus ou moins pointues et faciles d'utilisation. Par exemple, QtCreator intègre de la documentation sur les objets utilisés spécifiques à Qt, ce qui permet d'avoir rapidement accès aux différentes méthodes disponibles, et ainsi se passer d'un basculement fréquent entre navigateur et environnement de développement. Visual Studio propose un outil d'analyse de la mémoire très performant lors du débogage.

Afin de déboguer le programme, nous avons eu recours aux outils intégrés aux IDE, mais aussi à ddd, puisqu'il fonctionne pour des programmes C++. Nous avons fait ce choix quand nous n'avions pas encore développé de solution graphique. Nous utilisons alors à ce moment différents éditeurs de texte dotés de quelques outils facilitant le codage (snippets). Pour vérifier que le programme ne provoquait pas de fuite mémoire, nous avons utilisé le très reconnu valgrind jusqu'à obtenir un résultat satisfaisant (aucune fuite).

Enfin, toute la documentation de notre code a été produite en utilisant les raccourcis que propose l'outil Doxygen, qui est présenté dans les dossiers en annexe. La documentation est proposée en PDF (compilé avec LaTeX) et en version web HTML. Cela permet la production d'un document de qualité, quasi interactif, et très utile pour toute personne qui n'a pas connaissance du travail réalisé dans le cadre de ce TP. Nous y avons ajouté une liste des choses qui peuvent être à faire, cela permet de montrer les possibilités de cet outil, et donc la qualité de l'information (mise en relief des informations critiques...). Par exemple, l'ordre de présentation est bien pensé : les tâches restant à faire apparaissent avant la documentation, ce qui permet de renseigner le lecteur immédiatement, et ainsi lui éviter de mener une recherche, laquelle peut être longue et fastidieuse, au cœur de la documentation.

La gestion des versions de notre projet est réalisée à l'aide de Git, outil pour lequel des plug-ins sont intégrés à l'interface de développement.

Initialement, nous avons proposé le diagramme de Gantt suivant :



2 | Développement

Le développement du programme s'est fait en plusieurs étapes clef.

Dans un premier temps nous avons créé les différentes classes prévues dans notre diagramme de classes UML. Ce qui implique l'écriture des headers et également des constructeurs de base tels que nous les concevions à ce moment. Nous nous sommes chargés ensuite de mettre en place l'initialisation des planètes sur la grille du monde, pour cela nous nous sommes occupés de terminer le constructeur de l'objet world (qui n'a presque pas changé depuis). Enfin nous avons implémenté les méthodes dont nous étions sûrs de leur nécessité comme les getters et les setters de certains champs critiques, ainsi que des convertisseurs, puisque nous savions que les objets (agents) présents dans notre grille monde allaient devoir être supprimés puis remplacés à la volée durant la simulation. Par exemple nous avons créé une méthode permettant de « convertir » un objet free planet en un objet colonized planet en ne prenant en paramètre que des champs critiques. Dans ce cas précis, nous donnons en paramètre une référence vers la faction de la nouvelle planète colonisée. C'est également dans cette partie que nous avons créé la fonction scheduler qui parcourt les agents vivants et les fait agir. Cette partie que l'on pourrait qualifier de mise en place du système a duré pendant environ 20% du projet.

Après cette partie mise en place des agents dans la simulation, nous nous sommes attaqués aux méthodes run(), c'est-à-dire aux actions propres des agents dans la simulation que nous venions de mettre en place. Nous nous sommes donc d'abord chargés de la méthode run des planètes colonisées puisque c'est la plus complexe. Nous avons réalisé à ce moment-là que des méthodes run sur toutes les planètes n'étaient pas nécessaires, car il n'était pas prévu que les planètes sans faction agissent. Nous avons alors pu lancer nos premières simulations. Celles-ci nous ont révélé des erreurs non prévues, comme lors de la fin de la simulation qui renvoyait une segmentation fault à la suppression de la faction. S'en suivit donc une phase de bug fixing.

C'est notamment à ce moment-là que nous avons décidé de donner aux objets free planet une faction neutre pour avoir accès à plus de données. Comme celle-ci ne pouvait être créée qu'une seule fois (puisque en tant que faction fantôme, elle n'est pas un agent) nous avons décidé d'en faire un singleton. L'implémentation en elle-même des méthodes run() pour définir les actions des agents n'a duré que 5% du temps du projet mais la partie bug fixing a dû durer elle 60% du temps. En effet nous avons notamment rencontré des difficultés lors des conversions durant la simulation. Celles-ci rendaient les références des planètes voisines obsolètes. Cette mise à jour du voisinage fut la principale cause d'erreur durant le TP.

Enfin, une fois une version stable obtenue, et des simulations concluantes nous nous sommes lancés dans la création d'une interface en Qt avec un afficheur d'images à la place des caractères ASCII afin de découvrir les possibilités de l'outil. Pour ce faire nous avons utilisé très simplement l'outil en affichant des cercles pour chaque planète de couleur différente suivant la faction. Cette fonction display est appelée après chaque appel au scheduler. Nous avons également ajouté un tableau affichant les statistiques de chaque agent faction afin de voir en temps réel leur évolution. Nous avons ensuite peaufiné cet afficheur en rajoutant des images et des valeurs pour chaque planète (son économie et sa défense). Pour optimiser l'afficheur nous avons rajouté un booléen (qui n'était pas prévu à la base) sur les agents planètes déterminant si elle avait changé depuis le dernier tour (si ses valeurs ont évolué) et de ce fait si son affichage doit être mis à jour. Ce booléen est vrai à la construction des agents, ce qui fait que tout nouvel agent doit être affiché. De plus le getter de ce booléen passe automatiquement celui-ci à false en fin de traitement, de manière qu'une fois l'affichage mis à jour, celui-ci ne soit plus effectué tant que la méthode changed() (qui passe le booléen à true) n'a pas été appelée. Cette partie nous a permis de découvrir une erreur de référence concernant les factions puisque les valeurs affichées par le

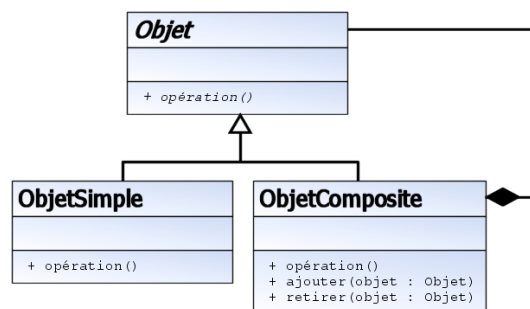
tableau étaient totalement erronées ce qui faussait également la simulation. Cette partie a duré 10% du temps du projet.

Pour terminer nous nous sommes chargés de modifier les valeurs de nos simulations afin de les rendre plus pertinentes. Nous avons également rajouté des constantes statiques dans nos classes permettant de contrôler plus efficacement les paramètres de notre simulation. Par exemple nous avons créé une constante `MAX_COLONY_DEFENSE` qui définit la valeur max de défense que peut avoir une planète. Enfin nous avons terminé le développement du projet par une phase de bug fixing qui nous a notamment permis de corriger des problèmes de fuite mémoire (à l'aide de valgrind). Nos agents planètes envoyant des demandes de subvention aux agents factions, ces demandes lors de leur traitement n'étaient pas supprimées ensuite, ce qui impliquait qu'au fur et à mesure des simulations, les factions perdaient inutilement de l'argent.

2.1 Patrons de conception

Durant le TP, 3 patrons de conception notables ont été utilisés :

2.1.1 Patron composite



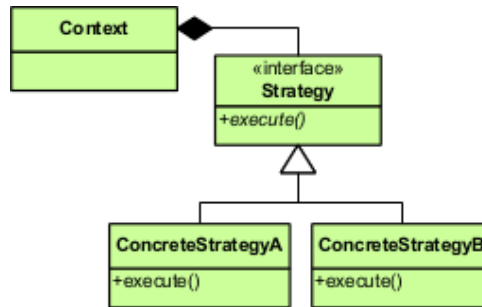
Dans notre code chaque agent planète a connaissance de ses voisins, sachant que ses voisins peuvent être soit des planètes libres, soit des planètes colonisées. Dans le code ce patron apparaît ainsi :

```

class Virtual_planet
{protected:
    std::vector<Virtual_planet* > neighbourhood_;
};
  
```

Ce patron nous évite bien sur de complexifier le code avec un classe voisin qui alourdirait grandement nos simulations.

2.1.2 Patron stratégie



Nous devons dans notre simulation traiter différemment nos agents en fonction de leur type, mais ceux-ci doivent être stockés sur une même grille monde. De ce fait nous avons été obligés de recourir au patron stratégie. Dans le code, celui-ci apparaît de la manière suivante :

```

bool Free_planet::is_attacked(Virtual_planet *) {
    bool res = true;
    if(world_.gen_mt(0,4)==0){ //Une chance sur 5 de ne pas réussir
        res=false;
    }
    return res;
}

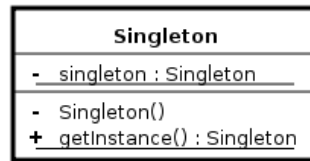
bool Colonized_planet::is_attacked(Virtual_planet *attacker) {
    bool res = true;
    if (attacker->get_faction() == this->get_faction()) {
        res = false;
    }

    if (colony_defense_ > world_.gen_mt(0,99)){ //plus la defense est grande, plus les
    ↪ chances de conquete sont reduite (defense <=50)
        res = false;
        colony_defense_ -= world_.gen_mt(1,25); //la defense et l'eco baisse a cause de l'assaut
        colony_production_ -= world_.gen_mt(1,25);
        change();
    }
    //On supprime l'agent de la liste d'attente puisqu'il va etre elimine ainsi que de sa faction
    if (res == true) {
        world_.remove_waiting_agent(this);
        faction_.remove_colony(this);
        faction_.remove_demand(this);
    }
    return res;
}

```

Ce patron nous permet donc pour une même instruction, d'obtenir un comportement différent en fonction de l'agent activé.

2.1.3 Singleton



Comme expliqué précédemment nous avons décidé de rajouter une faction neutre à notre code mais celle-ci ne devait pas peser dans la mémoire de nos simulations. Nous en avons donc fait un singleton :

```
class Neutral_faction: public Faction
{
private:
    static Neutral_faction* instance_;
    Neutral_faction(World& world);
public:
    static Neutral_faction* get_instance(World& world);
    static void dispose();
};
```

Il est à noter que pour terminer proprement le programme il faut appeler la méthode `dispose()` de ce singleton qui supprime proprement l'instance car le C++ ne vide pas automatiquement celle-ci à la fin du programme.

2.2 Présentation des interfaces

Nous avons développé deux interfaces durant notre projet, la première en ASCII, qui nous a notamment permis de debugger le programme. Nous avons également implémenté une interface graphique, plus agréable à regarder et plus ergonomique. Cette implémentation nous a permis entre autres d'apprendre les rudiments de Qt.

2.2.1 Interface console

L'affichage en mode console est une méthode de l'objet monde. Elle affiche dans la sortie standard chaque case de la grille du monde ainsi que les statistiques des factions. Son code est le suivant :

```
void World::display() {
    for (unsigned i = 0; i < len() + 2; i++) cout << "-";
    cout << endl;
    cout << "x|";
    for (unsigned i = 0; i < len(); i++) cout << /*"|"<<*/ i%10 <<"|";
    cout << endl;

    for (unsigned i = 0; i < hei(); i++) {
        cout << /*"|"<<*/ i%10 << " |";
```

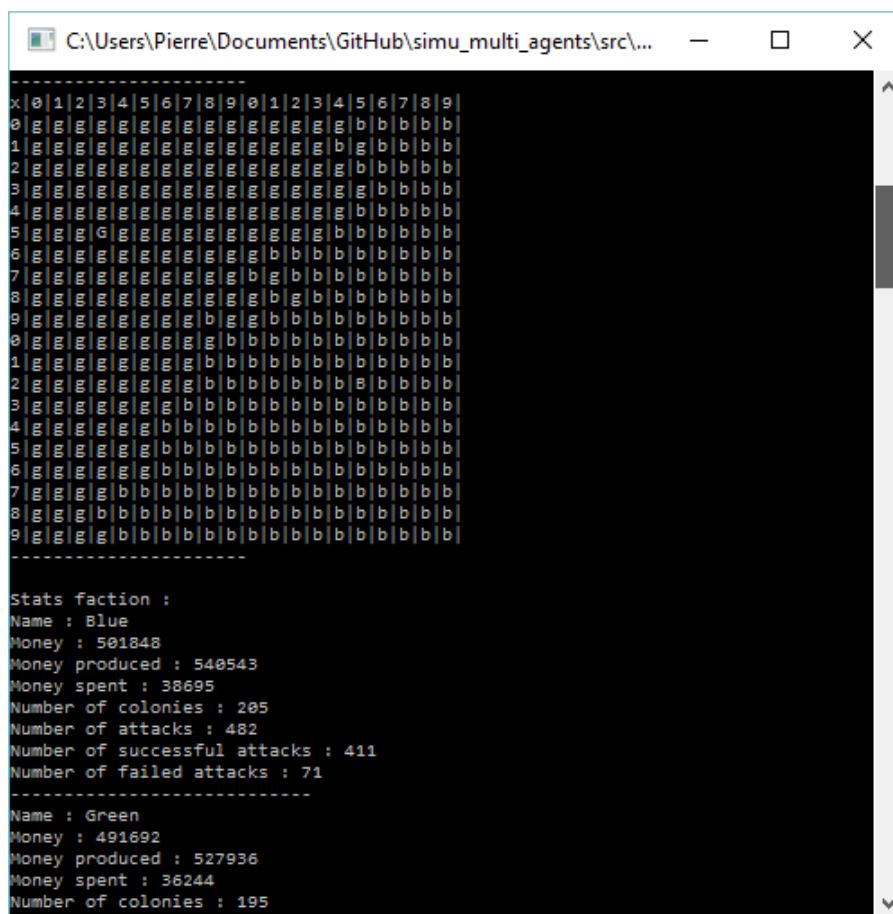
```

    for (unsigned j = 0; j < len(); j++) {
        cout << get_grid(j,i)->display();
        cout << "|";
    }
    cout << endl;
}
for (unsigned i = 0; i < len() + 2; i++)
cout << "-";
cout << endl << endl;

cout << "Stats faction:" << endl;
for(list<Faction>::iterator it = factions_.begin();
it!=factions_.end();
it++ ) {
    cout << it->toString();
}
cout << endl << endl;
}

```

Le principe est simple : on parcourt chaque case de la grille monde et on appelle la méthode `display()` qui est une fonction surchargée suivant le patron stratégie. Ainsi les factions ont un caractère ASCII représentant leur capitale et un représentant leurs colonies. Si on appelle la méthode `display()` sur une colonie le caractère correspondant sera donné etc. Si la planète est libre la méthode renverra un simple point. A l'écran cette interface apparaît ainsi :



```

-----
x|0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|
0|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
1|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
2|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
3|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
4|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
5|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
6|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
7|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
8|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
9|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
0|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
1|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
2|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
3|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
4|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
5|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
6|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
7|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
8|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
9|g|g|g|g|g|g|g|g|g|g|g|g|g|g|g|b|b|b|b|b|
-----
Stats faction :
Name : Blue
Money : 501848
Money produced : 540543
Money spent : 38695
Number of colonies : 205
Number of attacks : 482
Number of successful attacks : 411
Number of failed attacks : 71
-----
Name : Green
Money : 491692
Money produced : 527936
Money spent : 36244
Number of colonies : 195

```

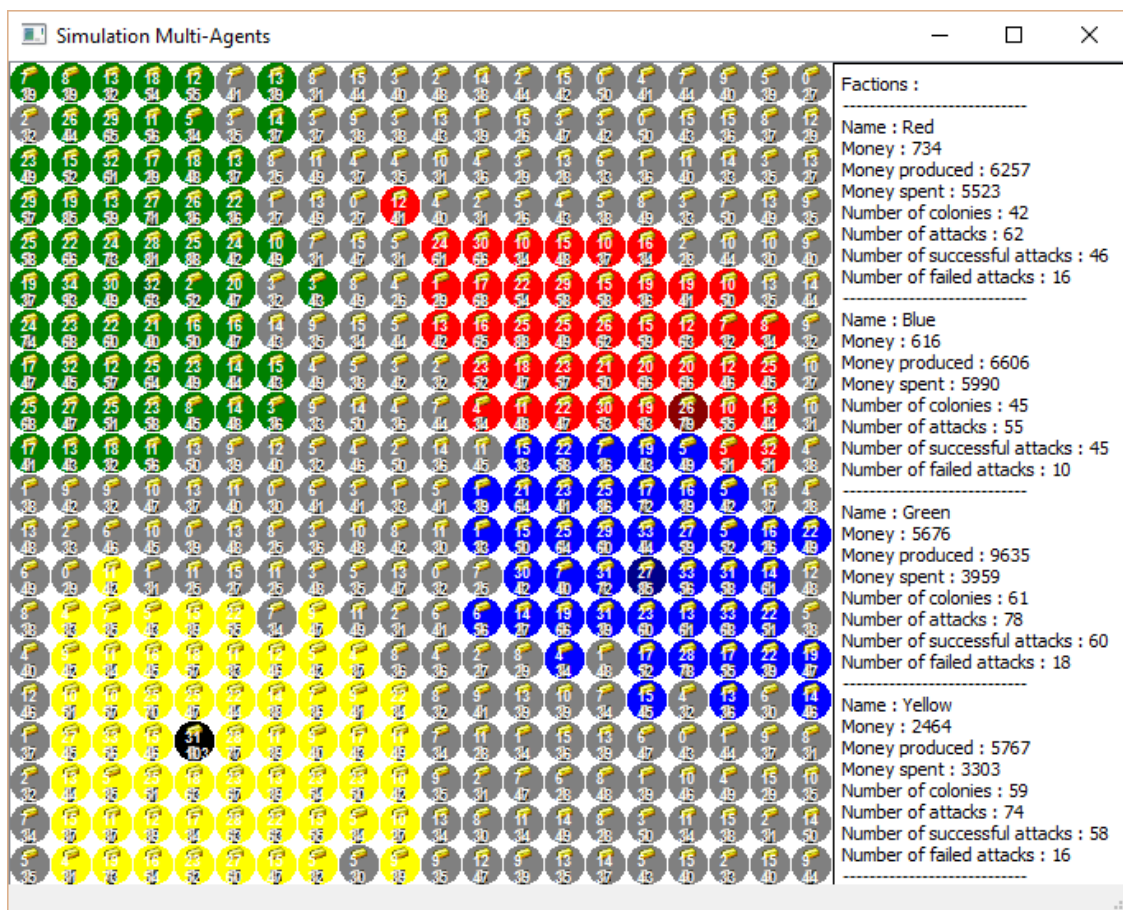
2.2.2 Interface Graphique

L'interface graphique faite sur Qt a pour avantage d'afficher plus de données que l'interface console et ce de manière plus claire. Elle possède elle aussi une fonction `display()` qui dessine les planètes et leurs statistiques quand celles-ci ont besoin d'être mises à jour (voir partie développement pour savoir comme l'afficheur détermine s'il doit redessiner une planète). L'afficheur graphique également met à jour les informations des factions en temps réel sur la droite de la fenêtre.

Il permet également de distinguer les factions à l'aide d'une couleur et non plus d'un caractère. Ceci est dû au fait que chaque faction possède des champs `color_colony` et `color_motherland` qui sont les couleurs (string) à donner aux fonctions de dessin de Qt pour obtenir certaines couleurs. Ainsi la fonction de Qt n'a qu'à appeler la méthode `get_color()` (qui est aussi surchargée) sur chaque agent planète et ainsi les couleurs affichées seront celles liées aux factions. Si la planète est libre sa couleur sera par défaut le gris.

Comme vu précédemment, un champ `change` nous permet de savoir si l'agent doit être redessiné. Si c'est le cas nous devons supprimer d'abord les anciens objets que nous avons dessinés. Pour cela, nous avons créé une classe `QPlanet`, composée de pointeurs vers une ellipse, deux images et deux textes, que nous créons et supprimons au besoin. Ainsi nous évitons les fuites mémoire.

A l'écran l'interface apparaît ainsi :



Le désavantage de cette interface est qu'elle ralentit beaucoup la simulation.

Source des images : <https://fr.wikipedia.org>

3 | Conclusion

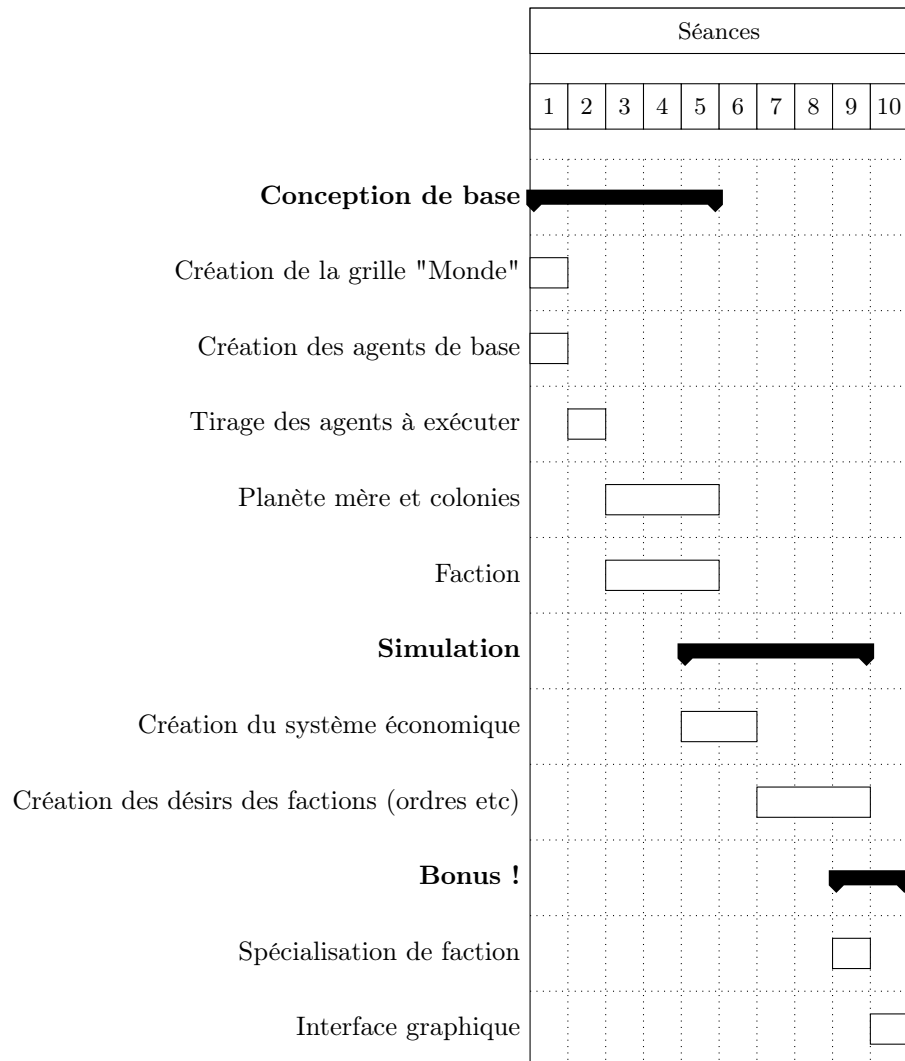
La réalisation de ce projet nous a permis d'acquérir plus de pratique du C++, et d'apprendre à penser un programme en équipe. Egaleme nt, la répartition du travail a eu son importance, et l'écriture d'un diagramme de Gantt nous a permis de jalonner toute la période de développement de différents repères afin de mener à bien la réalisation de ce projet, et de minimiser le nombre de bugs à corriger.

Nous sommes satisfaits de notre travail, même si nous aurions souhaité produire un programme encore plus abouti en termes de paramètres ajustables pour la simulation.

Nous avons été confrontés à différents problèmes lors du développement, notamment pour garantir l'"étanchéité" de la mémoire (Suppression du singleton, suppression des planètes remplacées lors des attaques...). D'autres problèmes se sont présentés lors de la suppression des planètes colonisées n'ayant pas encore joué, nous devions supprimer toutes leurs demandes de fonds pour empêcher des erreurs de segmentation qui arrivaient bien plus tard dans l'exécution et c'est pourquoi elles ont été difficiles à détecter. Nous avons à plusieurs reprises été victimes d'une erreur lors de suppression d'un objet pointé par un itérateur qui parcourait un conteneur. En effet, la méthode `erase(iterator)` retourne un itérateur sur l'objet suivant celui qui est supprimé dans le conteneur, or nous ne récupérions pas cette valeur, et ainsi cela provoquait une erreur de segmentation au tour de boucle suivant.

Nous avons été sensibles à la qualité du travail attendu, c'est pourquoi nous avons apporté beaucoup de soin dans le codage et la documentation de notre programme. Dans cette optique nous avons aussi proposé plusieurs exemples de simulation, à 2, 3 et 4 factions afin d'en observer le comportement et ainsi pouvoir ajuster certains paramètres (parfois borner certaines valeurs) de manière à obtenir des résultats plus réalistes.

Enfin, nous avons revu notre diagramme de Gantt initial pour finalement respecter celui présenté ci-après, plus réaliste compte-tenu de la quantité de travail à fournir.



Notre code source est disponible sur la plateforme GitHub à l'adresse :

https://github.com/theCmaker/simu_multi_agents

A | Utilisation de Git

Présentation réalisée par Pierre CHEVALIER.

A.1 Résumé

Git est un logiciel de gestion de versions décentralisé et libre développé notamment par Linus Torvalds et Junio Hamano. Il nous a permis de gérer les versions du projet conjointement avec le service d'hébergement de fichiers en ligne Github. Nous aurions également pu utiliser Git Server pour héberger nous-même les fichiers. Comme ce serveur aurait dû être lancé 24h/24 (car nous travaillons 24h/24) nous avons préféré utiliser une plateforme d'hébergement moins contraignante.

Les avantages notables de Git sont :

- Chaque objet enregistré possède un identifiant universel en SHA1 qui permet de l'identifier de manière unique,
- Git est multi-protocole, ce qui signifie que les échanges peuvent se faire en ssh, http(s) etc,
- Les objets stockés sont compressés,
- Pas de relation client/serveur,
- Gestion efficaces des branches de développement.

Il peut être utilisé directement depuis un terminal, ce que nous faisons lorsque nous travaillons sur les terminaux de l'ISIMA, mais peut aussi être utilisé par d'autre logiciel sous forme de plugin ou de programme dédié. Lors de développement du projet, nous avons notamment utilisé le logiciel de Github, Github Desktop, très utile dans le cadre d'une utilisation de Git sous Windows et non sous Linux ainsi que sous la forme d'un plugin intégré à Visual Studio.

Ces logiciels et plugin permettent une utilisation plus intuitive de Git, améliorant la vitesse de certaines actions prise par l'utilisateur.

Quelques mots de vocabulaire :

Version	On appelle version l'état du projet actuel, les fichiers présents, leur contenu etc. Git est un logiciel qui conserve toutes les versions enregistrées.
Repository	Le répertoire. Dans ce rapport nous appellerons repository le répertoire git composé de toutes les versions enregistrées du projet. Le repository local est l'arborescence de version présente sur le disque dur de l'utilisateur, le repository distant est l'arborescence présente sur le serveur git. Lors d'un push ou d'un pull (d'un envoi de version ou une récupération de la version distante), les deux repositories se synchronisent.
Commit	C'est un objet git (voir plus bas). On commit nos fichiers après une modification pour les enregistrer localement puis on les push pour envoyer les modifications au serveur.
Branche	Une branche est une version de l'arborescence du projet. C'est-à-dire une mutation du projet avec ces propres versions. Elle permet notamment d'éviter les conflits récurrents lorsque deux développeurs travaillent sur des fichiers communs. Cette pratique part du principe de préféré faire une grosse fusion de la branche à la fin du développement plutôt que de faire une multitude de petites fusions à chaque fois qu'un des développeurs poste sur le repository distant.
Clone	Un utilisateur de git peut cloner un repository. Git va copier toute l'arborescence du projet sur le disque dur de l'utilisateur. Celui-ci pourra ensuite envoyer ses propres commits sur le serveur distant s'il en a la permission. L'utilisateur peut aussi créer un nouveau repository en ligne basé sur cet ancien repository qu'il vient de cloner.
Index	L'index est la dernière modification enregistrée par git (le dernier commit). Quand il faut envoyer les commit au repository distant, git va d'abord vérifier que son index est à jour avant d'envoyer.
Conflit	Lorsque deux utilisateurs travaillent sur un même fichier, qu'ils le commit en même temps, il arrive que ces modifications soient issues d'une même version et donc lors de la mise à jour de repository distant (ou local si l'utilisateur n'a pas récupéré la dernière version enregistrée par le serveur distant) l'utilisateur peut être confronté à un conflit de fichier. Pour le résoudre, l'utilisateur peut choisir de conserver sa version locale ou la version distante ou encore faire une fusion des deux fichiers (il choisit quelles lignes conserver)

A.2 Procédure d'installation

Pour installer git sous un terminal il suffit d'utiliser la commande suivante dans un terminal :

```
$ apt-get install git
```

Si vous êtes sous Fedora, il faut utiliser la commande :

```
$ yum install git
```

Tentez alors de lancer la commande `$ git` pour vous assurer de la bonne installation du logiciel.

Si vous voulez également utiliser le service de cloud de Github, vous devez vous inscrire sur le site.

Pour installer Github Desktop il faut télécharger l'installateur à l'adresse suivante. Suivez les instructions et installer le logiciel. Celui-ci vous permettra d'installer de surcroît git sur votre ordinateur sous Windows en plus du logiciel Github Desktop.

Pour installer le plugin git sous Visual Studio il faut télécharger le plugin puis l'installer. Celui-ci vous permettra d'installer de surcroît git sur votre ordinateur sous Windows.

A.3 Présentation des fonctionnalités

Les logiciels de gestion de versions utilisent une arborescence de fichier permettant de conserver toutes les versions des fichiers et de mutualiser le développement entre les différents utilisateurs. Les développeurs mettent ainsi en ligne leur avancement accompagné d'un commentaire et le code source est mis à jour en fonction de cet avancement. Seuls les fichiers modifiés seront mis à jour. Un des avantages de ces logiciels est donc qu'ils limitent les doublons de données dans un projet qui peut être lourd en espace disque. Un autre avantage de ces logiciels est qu'ils permettent de gérer les conflits de version lorsque deux développeurs mettent leur avancement en ligne en même temps : une interface permet au dernier utilisateur qui a envoyé son travail de choisir quelles lignes garder.

Git se divise en deux structures de données : la base d'objets et le cache de répertoire. La base d'objet se trouve sur le disque dur client et se met à jour sur le serveur à chaque émission de versions et permet d'accéder à toutes les versions du projet alors que le cache de répertoire est uniquement sur la machine de l'utilisateur.

La base d'objet se compose de quatre types d'objet :

- L'objet blob (binary large objects) qui correspond à une version d'un fichier
- L'objet tree qui contient des objets blob comme un répertoire le ferait
- L'objet commit qui donne accès à une arborescence (à un tree racine et vers ses commit parents). Il possède également un message de log.
- L'objet tag qui est un message lié à un objet commit

Les actions possibles de l'utilisateur permettent d'ajouter et de supprimer ces objets.

Ces objets sont stockés dans le répertoire caché .git (situé dans le cache de répertoire). Le répertoire .git contient également des fichiers de configuration de l'environnement git, des informations sur les branches locales du repository, des logs, des infos sur l'état du prochain commit ...

A.3.1 Git dans le terminal

Le logiciel Git sous terminal consiste en une multitude de commandes permettant de créer des répertoires git, de récupérer les mises à jour distantes et d'exporter les mises à jour locales. Voici la liste des commandes les plus utilisées durant le projet et leur explication.

Commandes les plus utilisées :

- **git init** : Permet d'initialiser un repository sur le disque dur. Il va ainsi créer le .git d'un nouveau projet.
- **git clone** : Permet de créer une copie d'un repository existant en ligne. L'utilisateur a alors accès à ce repository et peut même faire des commit (si cet utilisateur est autorisé par le serveur)
- **git add** : Ajoute des fichiers de l'index du repository courant
- **git rm** : Supprime des fichiers de l'index du repository courant
- **git commit** : Prend tous les changements de l'index et crée un nouvel objet commit et sauvegarde cet objet en local. Un commentaire doit être associé à cet objet. Cette commande met à jour l'index du repository .
- **git status** : Montre les différences qu'il y a entre le répertoire réel et l'index de repository. Il va lister les fichiers non enregistrés dans l'index, les fichiers modifiés et les fichiers prêts à l'envoi (c'est-à-dire dont la version a été enregistrée dans un objet commit)

- **git branch** : Liste les branches existantes sur le projet. Crée une nouvelle branche si un nom est donné en argument.
- **git checkout** : Permet de changer de branche.
- **git merge** : Permet de fusionner deux branches. Cette action va automatique créer un nouveau commit s'il n'y a pas de conflits. Dans l'autre cas l'utilisateur va être invité à opérer à une fusion de fichiers ou à conserver une version.
- **git reset** : Permet de retourner à la version du dernier objet commit crée (version de l'index).
- **git pull** : Permet de récupérer les fichiers sur le repository distance vers le repository local. Equivalent à git fetch
- **git push** : Permet d'envoyer les objets commit locaux au repository distant.
- **git log** : Montre la liste des commit de la branche courante et les détails correspondants (les tags et les lignes modifiées par exemple)
- **git show** : Montre les détails des commits locaux

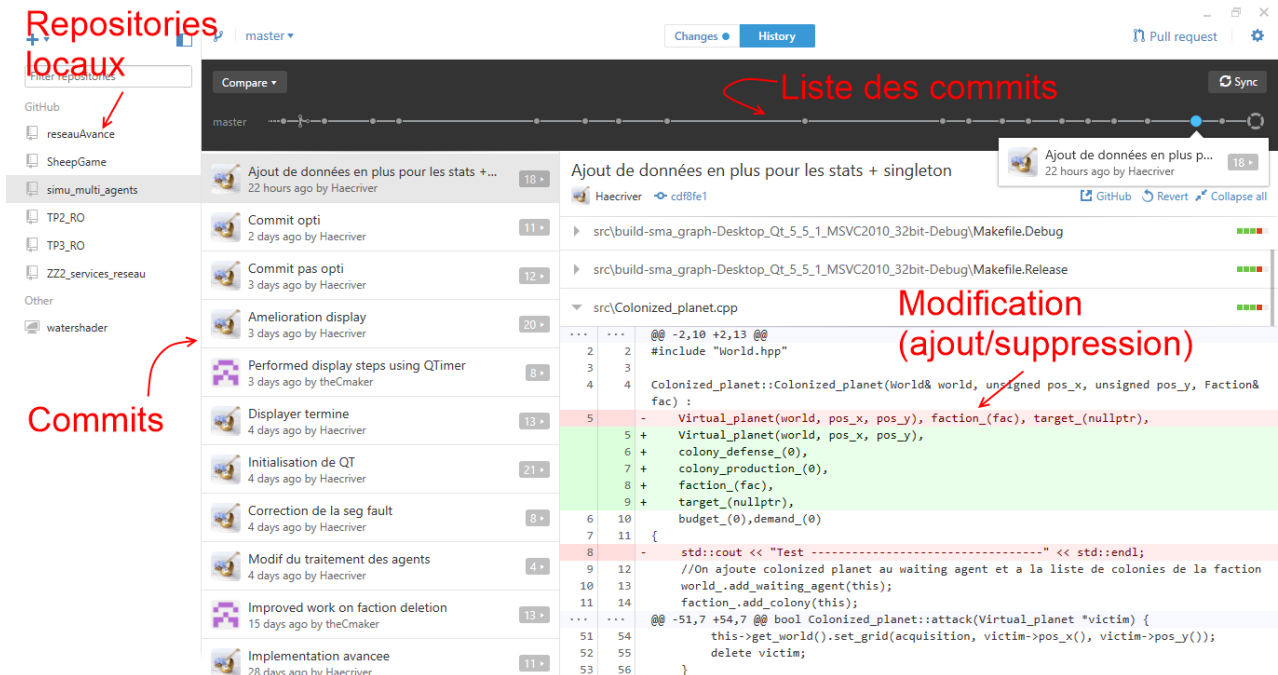
Autres commandes :

- **git config** : Permet de configure le nom, l'email, etc de l'utilisateur
- **git stash** : Sauvegarde des modifications que l'on ne veut pas commit immédiatement. Les changements pourront être appliqués plus tard.
- **git tag** : Lie un message (un objet Tag) à un commit.
- **git fetch** : Permet de récupérer les fichiers sur le repository distant vers le repository local
- **git remote** : Montre toutes les versions distantes du repository
- **git ls-tree** : Montre l'objet tree en incluant le nom de chaque objets et leur valeur en SHA-1
- **git cat-file** : Prend en argument la valeur en SHA-1 d'un objet et permet de voir un objet
- **git grep** : permet de faire une recherche à partir d'un mot clef dans un objet tree
- **git diff** : Affiche les différences entre les modifications de l'index et celle du répertoire courant

L'avantage de cette interface commande est qu'elle permet une grande marge de manœuvre à l'utilisateur puisque celui-ci à accès à l'ensemble des commandes git et donc à un contrôle total de son repository. L'inconvénient le manque d'intuitivité de certaine commande.

A.3.2 Git avec Github Desktop

Github Desktop est un logiciel fournit par Github mais qui peut également être utilisé sur des plateformes d'hébergement autres que Github puisque l'installation du logiciel donne aussi accès à une interface en ligne de commande de Git, donnant accès de ce fait à toutes les commandes Git. On peut ainsi cloner des repositories sur des serveurs externes à Github tout en profitant quand même de l'interface graphique.

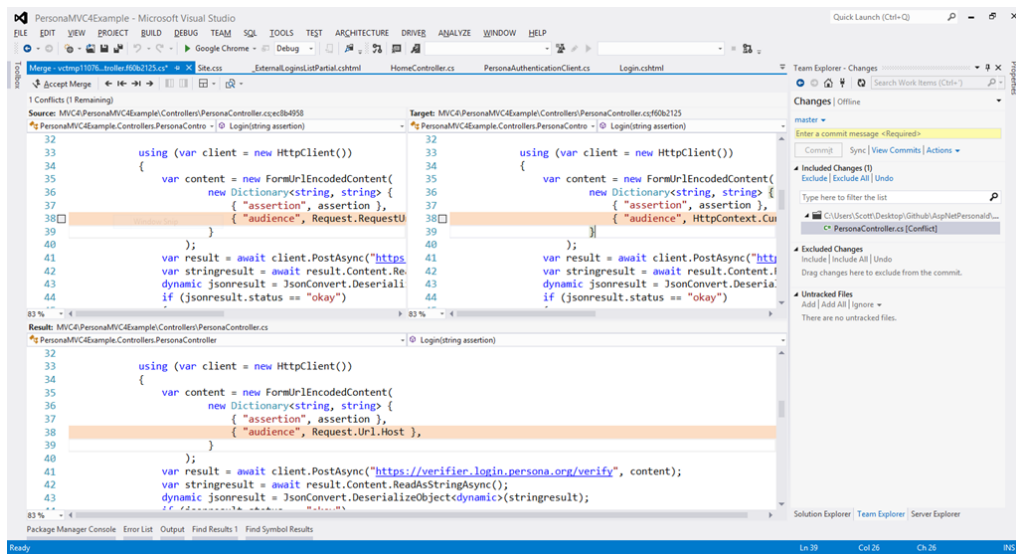


Celle-ci possède des capacités plus limitées que l'interface commande. Par exemple on ne peut pas cloner un directement depuis l'interface graphique un repository d'un serveur git distant (qui n'est pas Github) mais en contrepartie les merges sont plus intuitifs et la présentation des branches et commit est plus ergonomique.

A.3.3 Git avec Visual Studio

Le plugin de Visual Studio, tout comme le logiciel Github Desktop n'est pas aussi puissant que l'interface commande mais permet d'afficher l'historique des commits et surtout possède une interface facilitant grandement les merges si bien que nous n'avons pas eu besoin de travailler sur des branches durant le projet (sachant que celui-ci était réduit par rapport à de vrais projets professionnels bien entendu).

L'interface se présente ainsi :



Visual présente les deux versions en conflit avec une case à cocher pour savoir quelle version conserver. Le résultat est affiché en dessous des deux fenêtres. Un bouton permet de passer rapidement au conflit suivant. S'il y a un problème ambigu, l'utilisateur peut toujours modifier le résultat à la main.

Références :

<https://git-scm.com/>

<http://igm.univ-mlv.fr/~dr/XPOSE2008/git/fonctionnement.html>

<https://www.siteground.com/tutorials/git/commands.htm>

<https://fr.wikipedia.org/wiki/Git>

<https://fr.wikipedia.org/wiki/GitHub>

Image merge visual :

http://www.hanselman.com/blog/content/binary/Windows-Live-Writer/ed5b594103c4_B6D0/image_2.png

B | Doxygen – Présentation d'un outil de documentation

Présentation réalisée par Pierre-Loup PISSAVY.

B.1 Présentation

Doxygen est un outil permettant de générer automatiquement de la documentation à partir de code source. Ceci est possible grâce à l'implantation de balises particulières dans les commentaires présents dans le code. C'est un outil qui supporte plusieurs langages, et parmi les plus connus, le C/C++ et Java. La documentation ainsi produite peut alors être publiée, imprimée ou simplement sauvegardée.

Cela permet à la fois de produire de la documentation et de renseigner le développeur sur le code qu'il consulte. La documentation étant présente dans le code, cela permet de la garder à jour (un seul fichier à modifier pour mettre à jour la documentation), et facile à compléter.

B.2 Installation

B.2.1 Logiciel

Doxygen est d'une grande flexibilité, l'outil est disponible sur toutes les plateformes communes : Windows, Mac OS X, et les systèmes Unix (BSD, GNU/Linux).

Sur les systèmes Unix, il est présent dans les dépôts officiels de paquets orientés développement, on peut donc utiliser notre gestionnaire de paquets préféré, par exemple :

```
yum install doxygen
```

Pour les systèmes Windows et Mac OS X, on peut télécharger les installateurs sur le site officiel de Doxygen, <http://www.doxygen.org/download.html>, ou bien sur le dépôt GitHub officiel (<https://github.com/doxygen/doxygen>) si l'on souhaite obtenir ou tester les derniers changements.

B.2.2 Plug-ins

Doxygen est l'un des outils de documentation les plus connus, aussi l'on trouve de nombreux plugins afin de l'intégrer au sein des IDE les plus utilisés. C'est le notamment le cas d'Eclipse, NetBeans, QtCreator, MS VisualStudio et XCode.

B.3 Fonctionnalités proposées

Pour chaque document à produire, Doxygen nécessite un fichier de configuration, usuellement nommé *Doxyfile*¹. Ce fichier de configuration contient de très nombreux paramètres, et sa trame peut être générée automatiquement avec la commande

```
doxygen -g
```

Pour construire la documentation, on exécute simplement dans le dossier contenant le Doxyfile la commande suivante :

```
doxygen
```

Tous les paramètres de configuration sont comparables à des variables du shell et la syntaxe en est très proche.

Doxygen fait précéder chaque variable de commentaires expliquant son emploi et parfois les propriétés à respecter.

B.3.1 Réglage du rendu

PROJECT_NAME	Le nom du projet.
PROJECT_NUMBER	La version de la documentation ou du projet selon ce que l'on souhaite.
PROJECT_BRIEF	Une brève description du projet.
PROJECT_LOGO	Le chemin (relatif ou absolu par rapport au Doxyfile) d'une image ou logo représentant le projet (dimensions conseillées 200 × 50 px).
OUTPUT_DIRECTORY	Le dossier dans lequel la documentation doit être produite (celui du Doxyfile par défaut).
OUTPUT_LANGUAGE	La langue dans laquelle la documentation est produite (cela permet de faire correspondre le fond et la forme du résultat).

1. Par référence au Makefile de l'outil Make

B.3.2 Réglage de l'extraction de documentation

A l'exception des deux dernières, toutes les variables qui suivent sont booléennes et peuvent prendre la valeur YES ou NO.

EXTRACT_ALL	Impose à Doxygen de vérifier que toutes les classes ont été documentées.
EXTRACT_PRIVATE	Extrait la documentation des membres privés dans les classes.
EXTRACT_STATIC	Extrait la documentation des membres statiques.
EXTRACT_LOCAL_CLASSES	Extrait la documentation des classes imbriquées ou déclarées localement.
HIDE_UNDOC_MEMBERS	Enlève du document final les membres qui ne sont pas documentés.
RECURSIVE	Inspecte également la sous-arborescence à la recherche de fichiers comportant de la documentation.
EXCLUDE	Liste des fichiers et/ou répertoires à exclure de la recherche de documentation.
FILE_PATTERNS	Schémas des noms de fichiers qui peuvent être inspectés ² .

B.3.3 Et dans le code...

En C++, la fragmentation entre déclaration et définition pourrait présenter un frein à la documentation (où documenter, comment éviter les répétitions?), mais Doxygen reconnaît aussi bien les commentaires dans les headers que dans les fichiers source. Cela permet de documenter les classes, membres et méthodes simples (getters/setters) directement dans le header, et de documenter les méthodes et fonctions détaillées dans les fichiers source.

La documentation se fait donc dans des commentaires dont le formatage est spécial (il reprend les commentaires classiques du C++, en ajoutant simplement un caractère distinctif permettant de signaler le contenu comme de la documentation). Dans le cas du C/C++, cela peut-être un troisième caractère '/' pour les commentaires sur une ligne, ou bien l'ajout d'une étoile (*) sur la balise ouvrante des commentaires multiligne.

En voici un exemple :

```
/**
 * @class Personne
 * @brief Classe représentant une personne à l'aide de quelques informations de base
 *
 * Informations plus détaillées
 */
class Person {
private:
    std::string first_name_; ///< Prénom
    std::string family_name_; ///< Nom de famille
    Date        birthdate_;  ///< Date de naissance
    Gender       gender_;     ///< Sexe
};
```

Nous pouvons voir à travers cet exemple que les informations sont indiquées de manière sémantique. En effet, Doxygen nécessite l'emploi de balises afin de caractériser le type de l'information apportée.

2. Par défaut, Doxygen reconnaît les extensions usuelles, cette variable permet de sélectionner uniquement les extensions que l'on souhaite, et de les personnaliser

Ces balises sont généralement précédées indifféremment du symbole @ ou \.

Voici une liste non exhaustive des plus usitées :

@file	Nom du fichier.
@author	Auteur(s).
@details	Détails.
@version	Version du fichier.
@date	Date ou intervalle de dates fixes.
@bug	Pour signaler un comportement inattendu (permet de créer une liste de bugs).
@warning	Pour signaler les conditions d'utilisation à respecter.
@copyright	Licences.
@see	Référence vers un autre élément en relation avec celui portant la balise.
@deprecated	Élément obsolète.
@note	Information sur le comportement ou à destination des autres développeurs.
@param	Suivi du nom d'un paramètre et de sa description, pour décrire les paramètres d'une méthode/fonction.
@pre	Préconditions.
@post	Postconditions.
@remark	Remarques.
@return	Valeur retournée.
@since	Pour signaler depuis quand ou quelle version l'élément est disponible.
@todo	Choses restant à faire (une liste peut être constituée à partir de ces balises).
@struct	Description de structure en C/C++.
@class	Description de classe en C/C++.
@a	Suivi du nom d'un paramètre/argument, fait référence à un argument.

L'utilisation de Qt permet également de remplacer les symboles additionnels énoncés plus haut par des points d'exclamation (pour correspondre au style Qt). C'est ce que nous avons choisi dans notre projet (Qt-Creator pré-remplit les commentaires de documentation).

Dans les commentaires courts comme pour décrire le prénom, le chevron ouvrant remplace la balise @brief.

Enfin, il est possible de présenter la documentation de manière structurée en employant des listes de la même manière que dans un wiki, ou bien en utilisant directement du code HTML pour les compositions plus complexes.

B.4 Utilisation au sein du TP Multi-agents

Doxygen nous a permis d'obtenir une documentation dont la présentation est agréable et peut être rendue accessible à tous. Cela est possible en publiant la version web HTML ou bien en diffusant le PDF qui peut être généré à partir des fichiers source L^AT_EX produits par Doxygen.

Nous avons ajouté le logo de l'école et quelques informations sommaires sur le projet.

Il est bien évidemment possible de produire une documentation encore plus étoffée et consistante, mais cela demande toujours beaucoup de temps, tant du point de vue de la rédaction/mise en forme, que des revirements de situation dans le développement des objectifs à atteindre.

B.5 En bref

Pour résumer, Doxygen est un outil très complet permettant une documentation efficace et sémantique du code. Il présente le gros avantage d'être utilisable sur plusieurs plateformes, avec plusieurs langages et d'obtenir plusieurs formats de sortie, de la version Web aux pages de manuel en passant par l'incontournable PDF.

Sa renommée lui vaut l'existence de plug-ins pour les plus grands environnements de développement, ce qui n'est pas négligeable (auto-complétion, pré-documentation...) lorsqu'il s'agit de gagner du temps.

Les possibilités sont très étendues, et encore trop peu exploitées, c'est un outil qui mérite d'être promu auprès des développeurs (jeunes et senior) afin de rendre le développement et la maintenance de projets plus efficaces.

Sources d'information

- Site officiel de Doxygen :
<http://www.doxygen.org/>
- Initiation à Doxygen :
<http://franckh.developpez.com/tutoriels/outils/doxygen/>
- Using Doxygen :
<http://lugatgt.org/2002/05/30/using-doxygen/>
- Plug-in pour QtCreator :
<http://dev.kofee.org/projects/qtcreator-doxygen/wiki>
- 10 Minutes to document your code :
<http://www.codeproject.com/Articles/3528/Minutes-to-document-your-code>

C | Résultats

Ces résultats, bien que non utilisables puisque notre simulation a été créée avec des valeurs empiriques permettent toutefois de quantifier les valeurs et les probabilités liées aux agents de la simulation.

Les significations des cases sont les suivantes :

- Death : Nombre de tours avant la mort de la faction
- Money : Argent à la mort de la faction
- Prod : Argent produit par la faction
- Spent : Argent dépensé par la faction
- Nb colonies : Nombre de colonies à la mort de la faction
- Nb attacks : Nombre d'attaques exécutées par la faction
- Nb success : Nombre d'attaque réussies
- Nb fails : Nombre d'attaques échouées

- ML x : Position x de la planète mère de la faction
- ML y : Position y de la planète mère de la faction
- ML natdef : Défense naturelle de la planète mère (à l'initialisation)
- ML natprod : Production naturelle de la planète mère (à l'initialisation)
- ML findef : Défense finale de la planète mère
- ML finprod : Production finale de la planète mère

C.1 Résultats de cent simulations avec deux factions

C.1.1 Moyenne faction gagnante

Death	Money	Prod	Spent	Nb colonies	Nb attacks	Nb success	Nb fails
1028,34	156366,954	236889,458	180007,563	240,73	6251,58	5596,68	654,9

ML x	ML y	ML natdef	ML natprod	ML findef	ML finprod
8,96	9,47	38,16	8,27	89,56	25,39

C.1.2 Moyenne autres factions (éliminées)

Death	Money	Prod	Spent	Nb colonies	Nb attacks	Nb success	Nb fails
1026,34	131796,532	134702,934	156382,954	82,48	6099,82	5487,23	612,59

ML x	ML y	ML natdef	ML natprod	ML findef	ML finprod
9,29	10,01	37,98	6,54	88,55	23,79

C.2 Interprétation

En règle générale, il semble que les valeurs finales des factions gagnantes sont supérieures aux valeurs des factions vaincues.

Il est intéressant de noter que les défenses naturelles (défense à l'initialisation) des factions est un facteur qui a moins d'importance que la production naturelle (à l'initialisation) car si la moyenne du premier est environ égale à celle d'une faction gagnante contre une faction perdante, le deuxième lui est légèrement plus élevé pour la faction gagnante.

Un autre point important est qu'en moyenne la faction gagnante possède trois fois plus de colonies à la fin de la simulation que la faction perdante.