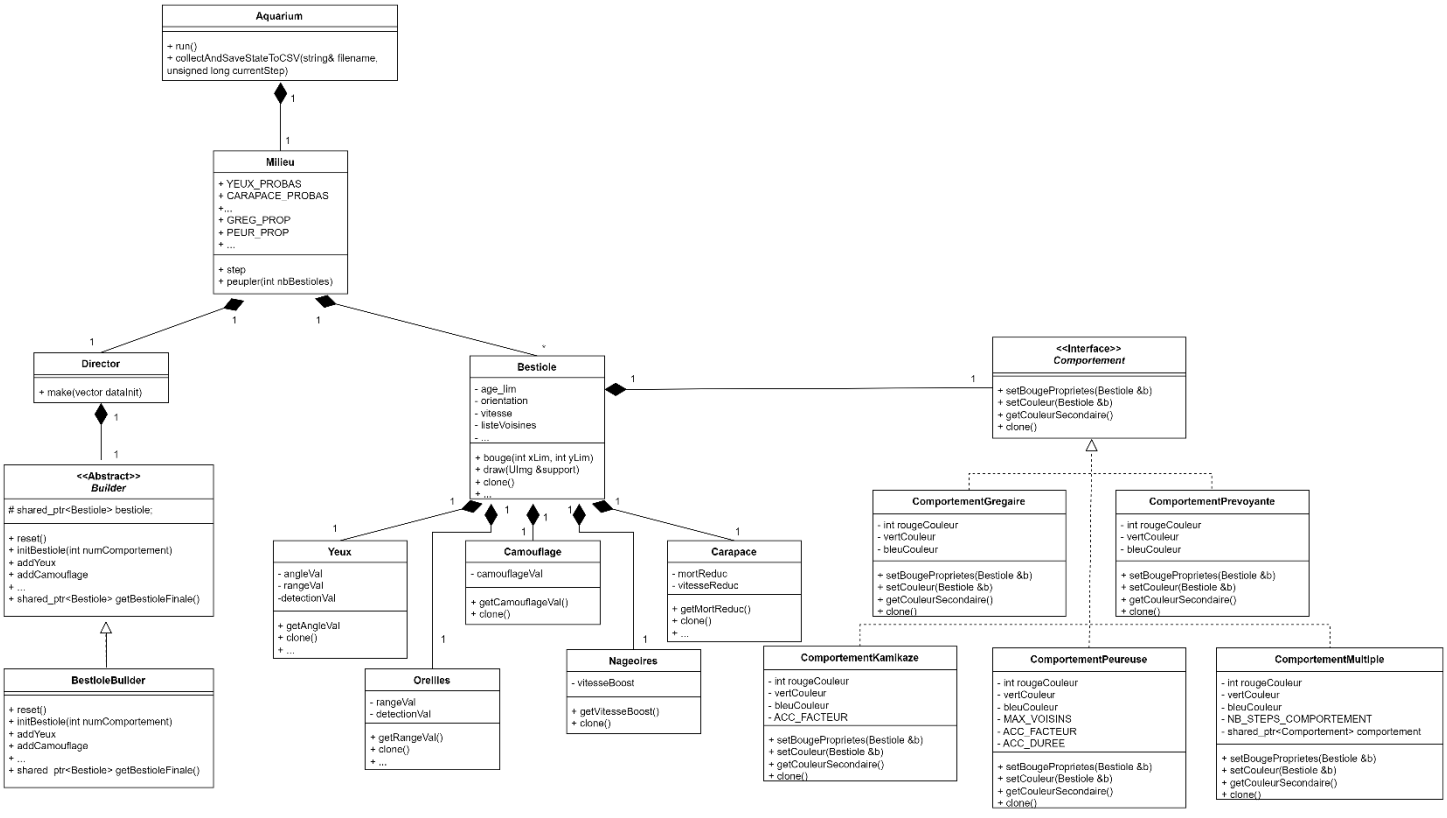
Compte-rendu : Etude de cas CPP

Ce compte-rendu regroupe quelques explications supplémentaires sur nos choix d'implémentation. La plupart des détails concrets sont accessibles dans le code mais ce document se veut être plus large. Vous y trouverez donc notamment nos quelques prises de liberté vis-à-vis de certains aspects laissés flous dans le cahier des charges ainsi que des commentaires sur les possibles améliorations ou évolutions de la version livrée.

I. Analyse objet du problème

Pour débuter notre analyse objet du sujet, nous avons étudié le fonctionnement de plusieurs design patterns pour voir lesquels pourraient s'appliquer à notre simulation. Si nous en avons initialement envisagés plusieurs (cf annexe présentation), nous n'avons finalement retenus que Builder, Strategy et Prototype qui seront détaillés plus bas.

1. Diagramme de classes



Pour éviter la surcharge, sur le diagramme de classes UML ci-dessus ne sont indiqués que les méthodes et attributs les plus essentiels ainsi que certaines constantes qui permettent de mettre en avant la configuration de la simulation. A noter que certains attributs essentiels sont représentés à travers les relations entre classe.

Voici un peu plus de détails sur le diagramme avec des explications allant du haut vers le bas du diagramme : Dans notre cadre d'étude, étant donné que nous ne lançons qu'une simulation à la fois, on considère qu'un Aquarium contient un unique Milieu et qu'un Milieu n'es

t associé qu'à un seul Aquarium. On estime également que le Milieu ne peut exister indépendamment de l'Aquarium.

L'Aquarium est en charge de la "simulation" à travers sa méthode run() qui gère, entre autres, l'interaction avec l'utilisateur. L'Aquarium a aussi la responsabilité de sauvegarder les données de la simulation dans un fichier CSV (plus à ce propos dans la suite).
Ensuite, un Milieu est composé de plusieurs bestioles (potentiellement aucune) et une bestiole ne peut appartenir qu'à un seul milieu à la fois. On suppose par ailleurs que l'existence des bestioles est intrinsèquement liée à celle de son milieu, comme un poisson ne pourrait vivre sans être maintenu dans l'eau par exemple.

Comme demandé, "la simulation dispose d'une unique ressource de création de bestioles", le Director. Le milieu lui passe les variables de décisions dans *dataInit* qui lui permettent de savoir le "type" de bestioles qu'il doit créer. (Nous avons choisi un *vector* pour contenir les *dataInit* pour avoir un accès direct via l'opérateur [ ] et ce en O(1)). Milieu a donc comme attribut *vector<unique_ptr<Bestiole>>* listeBestioles qui représente la liste des bestioles du milieu. Chaque élément de cette liste est un *unique_ptr* référençant une instance de Bestiole. Ces *unique_ptr* ont été obtenus via le Director et le BestioleBuilder. On a choisit des *unique_ptr* car on n'a pas vocation à accéder à une instance de Bestiole autrement que par cette liste listeBestioles et pour ne pas avoir à gérer la désallocation de la mémoire.

A chaque itération de notre simulation, on va changer les caractéristiques de chaque instance de Bestiole référencée par le *unique_ptr* de notre liste (par exemple la position). Toutefois, les bestioles possèdent un comportement qui change les caractéristiques d'une bestiole basée sur les caractéristiques de ses voisins. En itérant sur chacun des *unique_ptr<Bestiole>*, on va donc changer les caractéristiques de la bestiole référencée mais on ne peut pas se baser sur la liste listeBestioles pour déterminer les voisins de notre bestiole car listeBestiole aura déjà incorporé les modifications des bestioles faites avant la bestiole actuelle de l'itération. En effet, sinon cela voudrait dire que l'on n'update pas les bestioles simultanément : les dernières bestioles mises à jour auraient des informations plus actuelles sur les autres bestioles (et donc leurs voisins) que les premières bestioles mises à jour.

Pour empêcher cela, on va créer à chaque itération une nouvelle liste nommée listeBestioleAvantBouge. On va faire une copie indépendante de chaque bestiole référencée par un *unique_ptr* dans notre listeBestiole, pour sauvegarder les caractéristiques de la bestiole avant qu'on l'update. Ensuite, cette copie va être référencée par un *shared_ptr*, qui sera socké dans la liste listeBestioleAvantBouge.
On utilise un *shared_ptr* car cette listeBestioleAvantBouge va servir pour définir la liste des voisins pour chaque bestiole, une liste de voisins étant un *vector<shared_ptr<Bestiole>>*. En effet, une instance de bestiole peut être considérée comme voisine pour plusieurs bestioles, et *l'ownership* de cette bestiole « figée » avant update est donc partagé entre plusieurs pointeurs. De plus, l'utilisation de *shared_ptr* permet de réduire grandement le nombre de copies effectuées si nous n'en utilisions pas.]

Pour revenir sur la construction des bestioles, un directeur possède une instance concrète de Builder (évidemment, dans l'absolu, plusieurs instances sont possibles mais dans notre cas une seule est nécessaire). Vous remarquerez ici la mise en place du **Builder Design Pattern** (attention : on a fait un 'pseudo' Builder Design Pattern car on a pris une classe abstraite au lieu d'une interface et que le builder aurait dû garder un type générique dans le getFinale()). Nous avons opté pour ce choix afin de prévoir, comme souhaité, "la possibilité de générer par la suite de nouvelles espèces". En effet, on pourra facilement ajouter un nouveau builder, comme *AliensBuilder*. Certes, cette classe devra implémenter *addYeux()*, *addCarapace()*,... (bien que ces méthodes pourraient dans l'absolu être seulement dans BestioleBuilder afin de réduire le Builder au strict nécessaire) mais pourra avoir des méthodes en plus. Pour ce qui est de notre cas, nous nous sommes contentés

d'un BestioleBuilder qui permet de construire une bestiole de manière flexible. Pour cela, la classe possède un attribut bestiole (car en *protected* dans Builder) et peut y ajouter des accessoires et capteurs selon la demande du Director.
Une Bestiole possède un seul *unique_ptr* de chacun de ces accessoires et capteurs (potentiellement valant *nulltpr*). Nous avons considéré que ces derniers ne pouvaient exister indépendamment d'une bestiole et qu'ils sont associés à une unique bestiole. Nous avons utilisé des *unique_ptr* car il semblait plus judicieux que chaque bestiole possède ses *features* en propre. Et, de nouveau, des *smart pointers* ont été choisis pour ne pas se soucier de la désallocation de la mémoire (mais il fallait néanmoins veiller à bien faire des *deep copies* dans le constructeur par copie par exemple !).

Pour poursuivre sur nos choix d'implémentation des accessoires et capteurs, nous n'avons pas jugé nécessaire de définir une classe abstraite dont ils héritent car ils entraînent des changements assez différents et n'ont pas les mêmes attributs. En effet, utilisé le polymorphisme aurait été compliqué car l'incidence sur les propriétés (vitesse, probabilité de survie à une collision, ...) est assez différente selon les *features*. Ainsi, chaque *feature* a un ou plusieurs attributs se référant à leur "action" spécifique détaillée dans le cahier des charges. Chaque classe a aussi des constantes (non affichées sur le diagramme) afin que les valeurs min et max soient définies de manière commune à toutes les *features* d'une même sorte.

Autre attribut essentiel des bestioles : leur comportement (également un *unique_ptr* pour les mêmes raisons que ci-dessus). Ce comportement, à travers ses méthodes et attributs, impacte la manière d'agir et l'aspect physique des bestioles. Ainsi, les bestioles possèdent de nombreuses méthodes qui se basent sur leur comportement mais aussi sur leurs autres *features* pour modifier leurs propriétés physiques et comportementales. Par exemple, la méthode *bouge()* fait appel à la méthode *setBougePropriétés(Bestiole &b)* du comportement pour redéfinir l'orientation et la vitesse de la bestiole comme attendu. Nous avons ainsi utilisé le **Strategy Design Pattern** car les méthodes implémentées par les instances concrètes de Comportement sont interchangeables et Bestiole y fait appel en fonction de son comportement à travers l'interface Comportement.

On remarquera également une autre méthode importante pour le clonage spontané des bestioles : la méthode clone(). Cela nous permet de mettre en place la capacité des bestioles "à augmenter spontanément leur population par auto-clonage" en utilisant le **Prototype Design Pattern**. Il nous a semblé plutôt cohérent d'opter pour ce Design Pattern car il permet de facilement dupliquer un objet en donnant à cet objet la charge de son clonage. Etant donné que nous n'avons pas d'autres espèces, nous n'avons pas implémenter d'interface disons *Clonable* par exemple. En revanche, nous avons également utilisé le Prototype Design Pattern en l'intégrant à l'interface Comportement car là le polymorphisme est pleinement utilisé. (Nous avons ajouté une méthode clone() dans chaque *feature* également mais sans ici exploiter le polymorphisme car toutes les *features* sont assez différentes) Par ailleurs, pour être plus complet et poursuivre sur l'exemple d'*AliensBuilder*, nous aurions pu ajouter l'interface *Clonable* pour les espèces pour permettre à la classe *Aliens* de facilement se cloner tout en bénéficiant du polymorphisme en ne gérant que des pointeurs vers un *Clonable*.

Comme mentionné à l'instant, nous avons utilisé le polymorphisme pour Comportement. En effet, ne connaissant pas quel comportement aura chaque bestiole et ce comportement étant susceptible de changer, il était primordial de pouvoir manipuler des Comportements non concrets, notamment dans la classe Bestiole. Nous avons alors pu définir des classes concrètes qui implémentent l'interface Comportement. En plus d'avoir la méthode *setBougeProprietes(Bestiole &b)* mentionnée plus haut, un Comportement a une couleur spécifique qui permettra d'identifier visuellement le comportement d'une bestiole. Par ailleurs, vous aurez peut-être

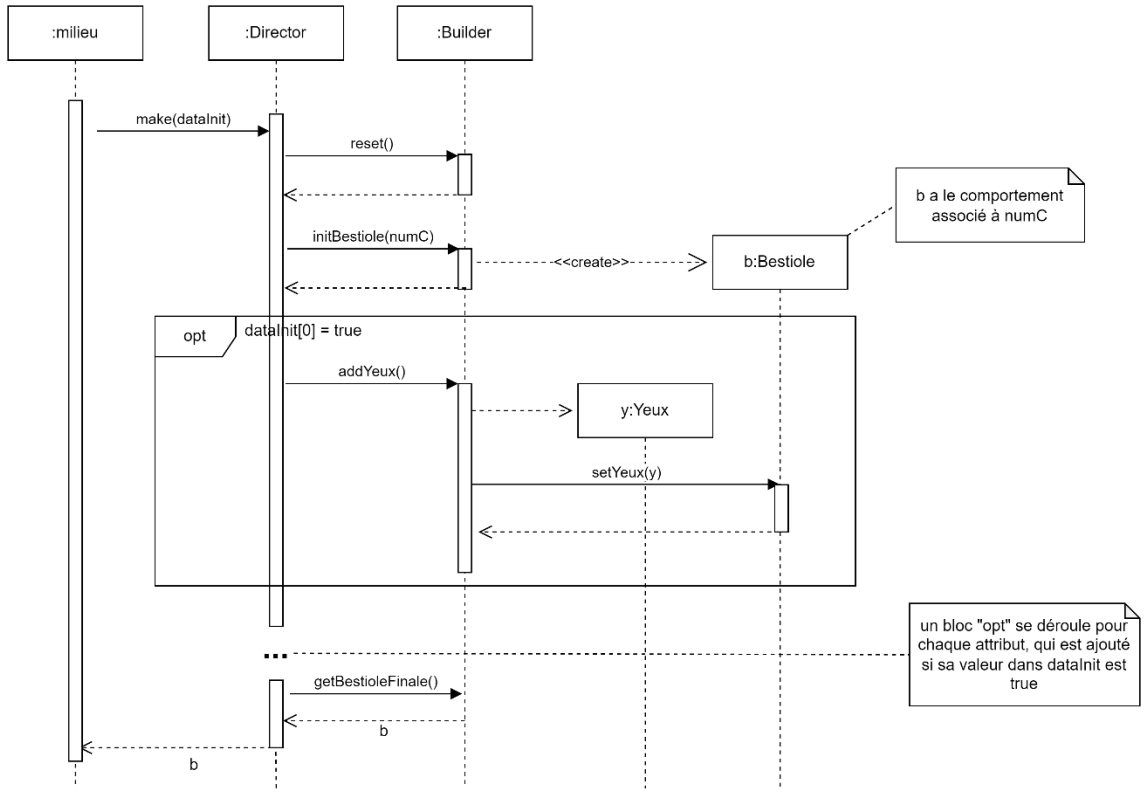
remarqué la méthode *getCouleurSecondaire()* qui peut paraître curieuse. Il s'agit ici d'une prise de liberté de notre part pour permettre de visualiser la manière d'agir d'une bestiole ayant un ComportementMultiple : la couleur de sa tête sera la couleur associée à son comportement sous-jacent.

Remarques :

- A toutes ces classes s'ajoute un fichier constantes.cpp (et son .h associé) qui nous permet de centraliser toutes les constantes citées comme faisant partie de la "configuration" possible de l'écosystème et souhaitée dans le cahier des charges. A noter que nous avons conservé les constantes en attribut de chaque classe, même si cela n'est dans l'absolu pas nécessaire. Ce choix a été fait pour permettre de mieux savoir, au sein d'une classe, ce qui est commun à toutes ses instances.
- Nous sommes repartis du code initial pour notre simulation mais avons modifié la représentation des couleurs des bestioles pour éviter d'avoir un pointeur et donc la nécessité de libérer la mémoire allouée et aussi pour éviter d'avoir des *casts* successifs (cf code).

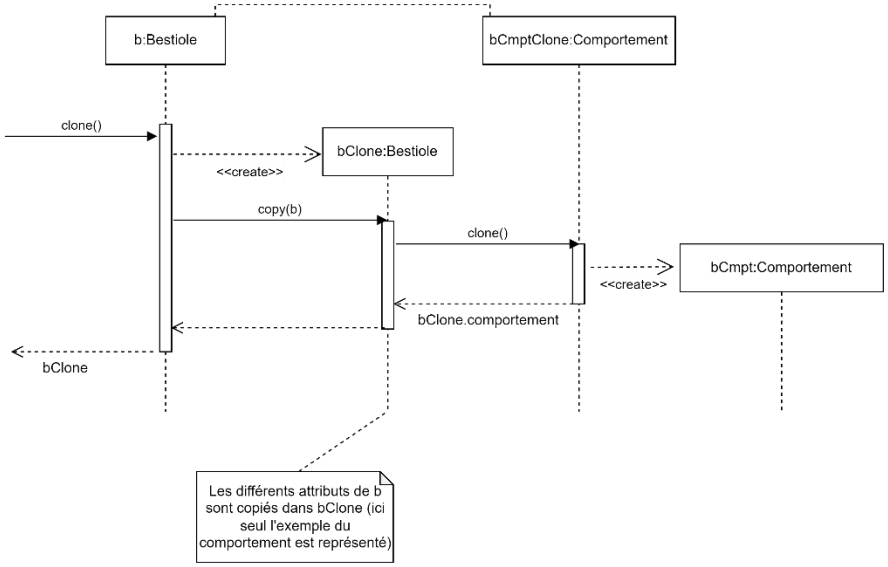
2. Diagramme de séquence

Construction des bestioles : Une des parties un peu plus complexes de notre code est la création de bestioles. Pour clarifier un peu son fonctionnement, voici le diagramme de séquence qui lui est associé :



Comme exigé, la création est centralisée. En effet, c'est Milieu qui a connaissance de toutes les probabilités d'avoir telle ou telle *feature* et les proportions des comportements. Cela est traduit sous forme de variables de décisions dans *dataInit* qui est passé au Director. C'est donc à lui qu'est déléguée la gestion de la création des bestioles. Ce dernier fait aussi appel à son builder pour lui renvoyer une bestiole avec les caractéristiques demandées. La fonction make() est ainsi exploitée au sein d'une boucle pour peupler l'aquarium avec le nombre de bestioles souhaitées initialement. Mais on y fait aussi appel pour la création spontanée et l'ajout lors d'un évènement extérieur au sein de la méthode naissanceBestiole() (méthode dans laquelle on transcrit les proportions de comportements souhaités en probabilités).

Clonage : lors d'un clonage spontanée d'une bestiole (qui a lieu avec une certaine probabilité à chaque étape), on fait appel à la méthode clone() qui appel le constructeur par copie de Bestiole qui lui-même fait appel à la méthode clone() des *features* et du comportement de la bestiole d'origine :



II. Implémentation

Concernant l'implémentation des différentes fonctionnalités, l'ensemble des demandes a pu être réalisé. Vous trouverez de plus amples explications du fonctionnement concret au sein du code mais voici un rapide aperçu de ce qui a été fait :

- **Naissance** : le milieu centralise la création de bestioles à travers ses méthodes *peupler(int nbBestioles)* et *naissanceBestiole()*. Cette seconde méthode permet la naissance spontanée d'une bestiole (possible selon une certaine probabilité configurable dans Constantes.cpp) à chaque step. On peut également ajouter une nouvelle bestiole en appuyant sur 'a' ou 'A' durant la simulation.

- **Mort** : les précisions du cahier des charges ont été implémentées et toutes les variables sont modifiables dans Constantes.cpp. Concernant la destruction d'une bestiole par un évènement extérieur, il se fait en appuyant sur 'm' ou 'M' et en saisissant l'identifiant de la bestiole qu'on souhaite achever.

- **Clonage** : expliqué plus haut et la probabilité associée est commune à toutes les bestioles (et configurable dans Constantes.cpp : pour éviter les répétitions, vous pouvez à présent partir du principe que tout ce qui a été mentionné comme paramétrable dans le cahier des charges est disponible dans Constantes.cpp.)
  A noter que les bestioles « se subdivisent » littéralement : le clone part de la même position que la bestiole qui en est à l'origine. En conséquence, pour des kamikazes, il y a de fortes chances que la bestiole et son clone meurent car elles s'attaquent mutuellement. Cela pourrait être éviter en faisant naître les clones (de kamikazes en particulier) à une position aléatoire.

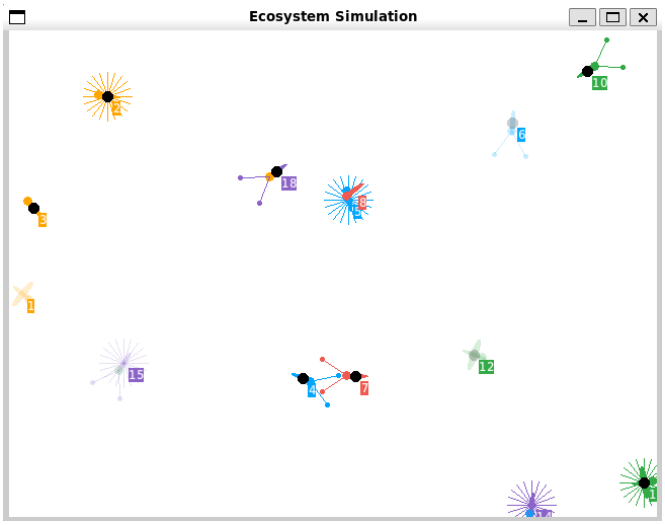
- **Capteurs et accessoires** : les valeurs caractéristiques (angle, rayon, etc…) de chaque *feature* est initialisé aléatoirement en utilisant une distribution uniforme entre les valeurs min et max.

- **Détection** : a été centralisée dans la méthode jeTeDetecte.

- **Comportement dynamique** : leur fonctionnement détaillé est disponible au sein du code. Néanmoins, voici quelques précisions sur nos choix d'implémentation qui ne sont pas abordées dans le code :
    - Les grégaires prennent en compte la vitesse et l'orientation moyenne des bestioles environnantes.
    - Les peureuses fuient durant une certaine durée et à une certaine vitesse configurable.
    - Pour les kamikazes, nous avons fait en sorte qu'une vrai « attaque » soit mise en place. Nous avons donc décidé que les kamikazes accélèrent (exigence bonus suite à notre discussion avec M. Guériot !) en direction de la bestiole la plus proche. Au début nous avions mis une accélération rapide vers la bestiole la plus proche mais il était complexe de calibrer cette vitesse. A la place, nous avons donc fait en sorte que les kamikazes accélèrent en direction de la bestiole la plus proche tant qu'elle est encore dans son champ de vision. Une version encore plus poussée serait de définir un attribut bestioleCible et de la suivre / accélérer vers elle pendant une durée plus longue et non plus contrainte par le champ de vision. Cela supposerait donc un réel *tracking* des autres bestioles et donc plus seulement la connaissance des bestioles voisines.
    - Pour les prévoyantes, elles se dirigent vers l'opposé de la moyenne des bestioles environnantes. Ce choix est contraignant au niveau des frontières où elles risquent de rester bloquées. Une amélioration pourrait être de longer la frontière jusqu'à ce qu'il n'y ait plus de voisins dans leur champ de vision (même possibilité pour les peureuses). Par ailleurs, une version qui utiliserait de nouveau le *tracking* serait de prévoir en amont si des bestioles se dirigent vers l'opposé de la moyenne des bestioles environnantes. Cela éviterait de rechanger de trajectoire en calculant directement la direction de « fuite » optimale.
    - Pour les personnalités multiples, nous avons décidé qu'elles changeront toutes en même temps de comportement (mais celui-ci est aléatoire et donc potentiellement différent) et au bout d'un temps fixé au début de la simulation. Pour aller plus loin nous aurions pu rendre ce changement plus aléatoire en définissant également le temps avant un changement de manière aléatoire. Par ailleurs, nous aurions pu mieux adapter le changement de comportement en s'assurant que la bestiole retrouve sa vitesse initiale et ne conserve pas la vitesse acquise lors d'une accélération (pour peureuse ou kamikaze par exemple).

  Le changement de comportement d'une bestiole peut se faire en appuyant sur 'c' ou 'C' au cours de la simulation et en saisissant l'identifiant de la bestiole dont on souhaite modifier le comportement. Le nouveau comportement est déterminé aléatoirement.

- **IHM** : au cours de la simulation, l'utilisateur peut déclencher 3 évènements extérieurs (mentionnés plus haut). Par ailleurs l'appui sur toute autre touche affiche les identifiants (par exemple on peut appuyer longtemps sur « espace » pour les voir).



Finalement la touche « Echap » arrête la simulation.
(L'aspect physique choisi pour visualiser les comportements et les *features* est disponible dans la présentation.)

- **Suivi de la simulation** : Pour pouvoir analyser le déroulement de la simulation, nous avons extrait le nombre de bestioles par comportement et pour chaque *feature* dans un fichier CSV. Nous avons également extrait les constantes dans un autre fichier CSV. Ces deux fichiers ont pour suffixe la date et l'heure de début de simulation afin que leurs données puissent facilement être recoupées par la suite.

III. Tests

Pour tester le bon fonctionnement de nos méthodes, nous les testions une par une (que ce soit pour les comportements ou les *features*), à la fois visuellement et en affichant dans l'invite de commandes les informations pertinentes.
A présent que la phase développement est terminée, les tests sont réalisables en observant la simulation et en modifiant la configuration initiale.

1. Comportements

Ainsi, commençons par des tests « unitaires » pour voir que l'impact du comportement sur la durée de vie **d'un groupe de 10 bestioles** avec toutes des capteurs de mêmes propriétés et sans accessoire. Ces tests sont dits unitaires car les parties aléatoires ont été suspendues (pas de naissance ni d'évènement extérieur utilisé) et seule la mort par collision est maintenue. Par ailleurs nous avons fait les tests une dizaine de fois pour obtenir une moyenne et attendu 30 secondes (car sinon on aurait pu attendre longtemps que toutes les bestioles meurent…).

Comportement	Nombre moyen de bestioles vivantes au bout de 30s
Grégaire	3,2
Peureuse	3,4
Kamikaze	1,2
Prévoyante	9,8
Multiple	2,7

Il est donc assez satisfaisant de voir que dans un groupe de kamikazes il y a nettement moins de chances de survie tandis qu'un groupe de prévoyantes reste quasi toujours au complet. Par ailleurs, il ne semble pas très favorable pour les bestioles d'être mélangées entre elles comme le montre le résultat pour personnalités multiples. Finalement, les grégaires et peureuses ont des chances de survies assez semblables : elles finissent par faire le même mouvement en restant proches pour les grégaires tandis que les peureuses peuvent facilement se déplacer sans risquer d'entrer en collision lors de leur fuite quand elles ne sont plus que 3 ou 4.

2. Accessoires et capteurs

Testons le même scénario qu'avant avec des kamikazes mais en leur ajoutant des accessoires puis un seul des 2 capteurs :

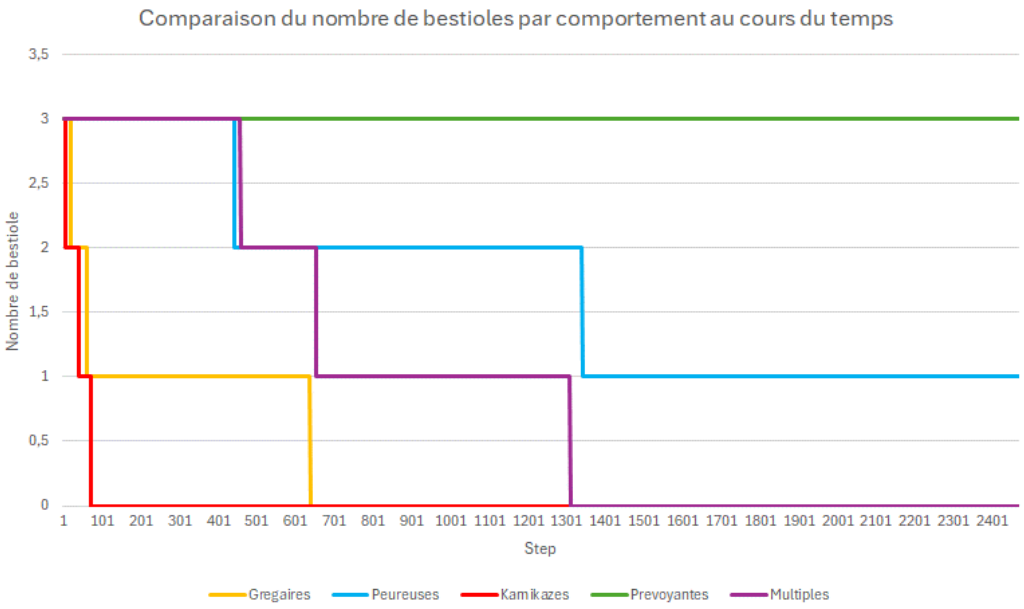
Accessoire	Nombre moyen de bestioles vivantes au bout de 30s		Capteur	Nombre moyen de bestioles vivantes au bout de 30s
Nageoires	1,4		Yeux seuls	1,9
Carapace	1,6		Oreilles seules	1,3
Camouflage	2,8			

On observe donc que les accessoires permettent bien d'augmenter l'espérance de vie des bestioles (surtout le camouflage) et que les yeux permettent une moins bonne détection que les oreilles (ce qui entraîne donc moins d'attaque et une augmentation du nombre de survivantes).

(NB : la suite des validations est consultable dans le diaporama de la présentation.)

IV. Résultats

Un énorme nombre de scénarios est possible. Pour faire simple, on peut tout d'abord essayer de mettre toutes les bestioles de comportement différents ensemble et ce dans des proportions égales.
Sans surprise, les kamikazes sont les premières à disparaître et les prévoyantes sont celles qui survivent au groupe.



Plus de scénarios seront présentés au cours de la présentation.

V. Gestion du travail collectif

Concernant la gestion du travail collectif, nous avons utilisé GitHub afin de centraliser le code et de facilement collaborer. Nous avons divisé le codage de la simulation en différents sous thèmes et chacun avait la charge de plusieurs de ses sous thèmes. Nous restions néanmoins très régulièrement en contact (quand nous codions en dehors des horaires de cours) afin de se mettre d'accord sur les types par exemple pour faciliter la collaboration mais aussi pour prendre la décision la plus adaptée. La subdivision en sous thèmes nous a permis de travailler en parallèle et donc d'être plus efficace. La communication a été très présente et primordiale pour notre réussite tout comme l'entraide pour la résolution de problèmes techniques.