Projet Logiciel Transversal

Élevage Numérique

M. GOSSELIN

Sommaire

I. Présentation du logiciel

Nom du logiciel : Élevage Numérique
Principe du logiciel : Le logiciel Élevage Numérique est un logiciel de gestion d'élevage ou de populations précises d'animaux.

Fonctionnement du logiciel

Un fermier (ou même un zoo), élève des animaux d'une race bien précise.
Il doit donc pouvoir gérer la population d'animaux de cette race, les nourrir, les vendre, les faire s'accoupler en évitant un maximum les problèmes de consanguinité, acheter d'autres animaux, gérer les frais vétérinaires, tuer les bêtes malades ou âgées, tout en assurant un gain maximal (pour un zoo, il n'y a évidemment pas de gain possible mais que des dépenses).

À l'initialisation du logiciel, il faut créer une nouvelle population d'animaux. Il est possible de créer plusieurs populations d'animaux.
Quand on créée une population d'animaux, il faut renseigner :
-la race animale élevée,
-les animaux qui appartiennent à cet élevage.

Si la race de cet élevage, n'a jamais été élevée dans la ferme, il faut alors créer une fiche de renseignements sur la race en question, comprenant :
-le nombre moyen de petits qui naissent dans une portée,
-le poids moyen d'un mâle et d'une femelle adultes de cette race,
-l'espérance de vie de l'espèce (pour le mâle et la femelle),
-l'âge moyen de maturité de cette race,
-la durée moyenne de gestation de la race,
-le prix d'une bête à la vente (pour une bête enfant, et adulte)
-le prix d'une ration de nourriture par unité de temps pour chaque bête,
-le nombre N de générations minimal pour que la présence d'un ancêtre commun entre deux individus de sexes différents ne créée pas de réel problème génétique, dans le cas d'accouplement.

Ensuite, pour chaque individu de cette espèce animale, il faut renseigner l'âge et le sexe. Un numéro est attribué à chaque animal automatiquement.

L'éleveur peut décider à tout instant de vendre ou tuer un animal, d'acheter un nouvel animal pour enrichir le patrimoine génétique du cheptel, ou de faire se reproduire des animaux de sexes opposés (si les deux bêtes à accoupler ont un ancêtre commun, l'accouplement est impossible).
Quand deux animaux s'accouplent, les animaux résultant de cet accouplement, sont automatiquement ajoutés à la base de donnée, au bout de la durée de gestation (dans un but de simplification, le nombre d'animaux qui naissent est systématiquement le même et correspond toujours au nombre moyen d'enfants par portée pour cette race animale, le sexe des animaux nés, est attribué aléatoirement ou défini manuellement).

Quand tous les animaux d'un élevage sont tous supprimés de la base de donnée (tous vendus ou morts), l'élevage est automatiquement supprimé du logiciel.

Il est possible de créer un élevage d'une espèce déjà exploitée dans le passé, il n'est alors pas nécessaire de créer une fiche de renseignements pour cette race.

L’unité de temps est le mois.

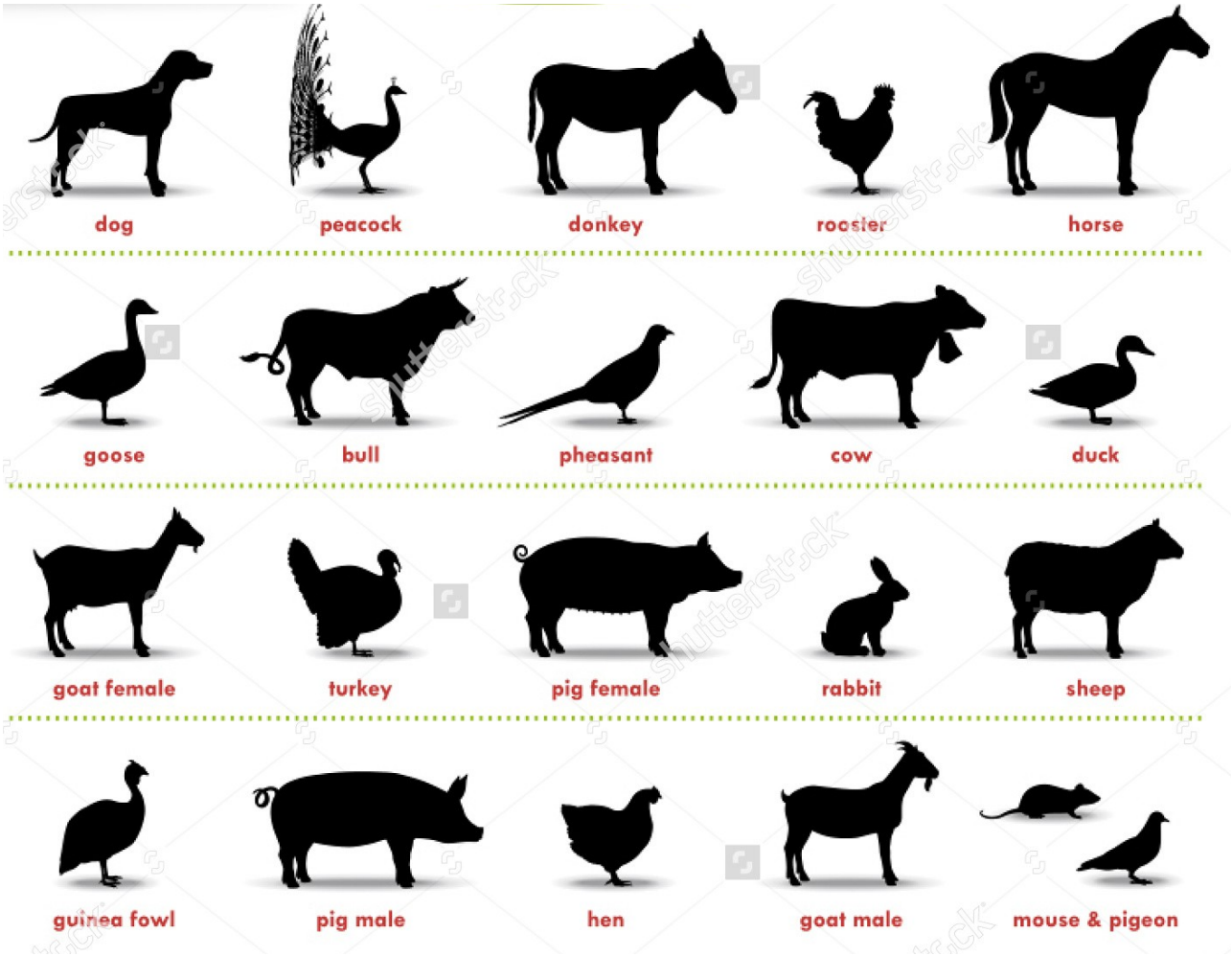
Le logiciel doit pouvoir gérer au moins 10 espèces animales différentes composées au minimum chacune de 500 individus.

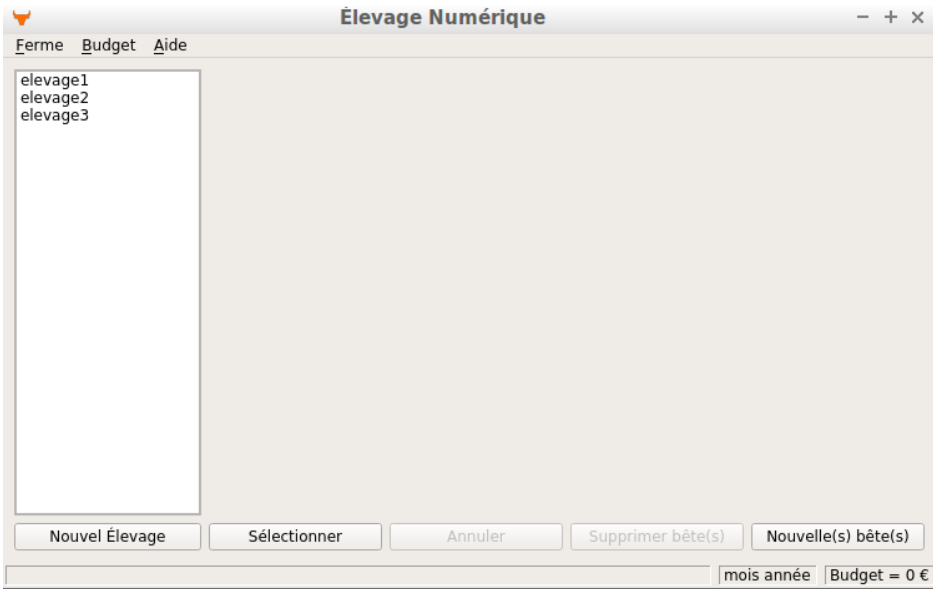
Pour simplifier les données :
On considère au début, que les animaux ont tous le même âge et n’ont aucun ancêtres communs (donc qu’on peut commencer à reproduire les animaux ensembles dès le lancement du programme), et qu’il n’y a aucune différence de poids moyen adultes et d’espérance de vie, entre mâles et femelles.

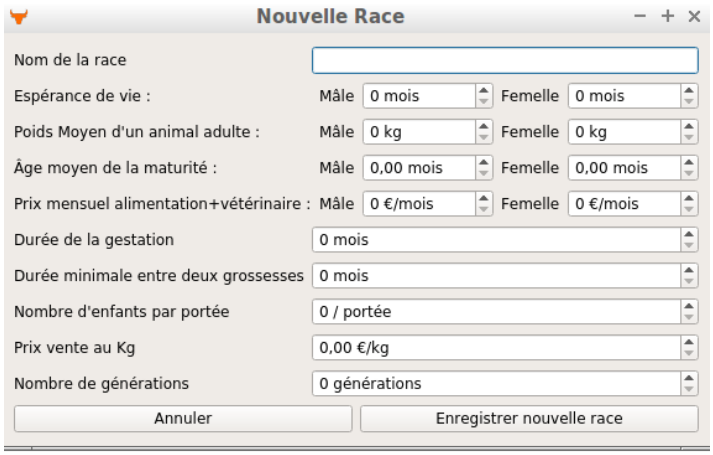
Environnement de jeu

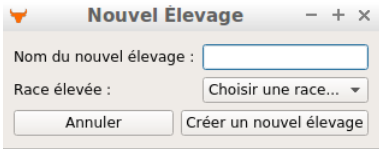
L’environnement de développement du logiciel est NetBeans IDE.

Ressources




Élevage Numérique
Ferme Budget Aide
elevage1
elevage2
elevage3
Nouvel Élevage
Sélectionner
Annuler
Supprimer bête(s)
Nouvelle(s) bête(s)
mois année
Budget = 0 €


Nouvelle Race
Nom de la race
Espérance de vie :
Mâle 0 mois
Femelle 0 mois
Poids Moyen d'un animal adulte :
Mâle 0 kg
Femelle 0 kg
Âge moyen de la maturité :
Mâle 0,00 mois
Femelle 0,00 mois
Prix mensuel alimentation+vétérinaire :
Mâle 0 €/mois
Femelle 0 €/mois
Durée de la gestation
0 mois
Durée minimale entre deux grossesses
0 mois
Nombre d'enfants par portée
0 / portée
Prix vente au Kg
0,00 €/kg
Nombre de générations
0 générations
Annuler
Enregistrer nouvelle race


Nouvel Élevage
Nom du nouvel élevage :
Race élevée :
Choisir une race...
Annuler
Créer un nouvel élevage

II. États du programme

Description des états, Conception logiciel[1]

Le logiciel présenté ici sert à une Ferme. La classe **Ferme** est caractérisée par les élevages qui la composent, les espèces d'animaux élevés dans ces élevages, et son budget.

La classe **Elevage** est un ensemble d'animaux d'une même espèce. Chaque élevage qui compose la ferme est caractérisé par un identifiant, une espèce animale (ou race) à laquelle appartient tous les animaux de l'élevage, et par les animaux qui composent l'élevage.
Chaque élevage est caractérisé par un état : Actif ou

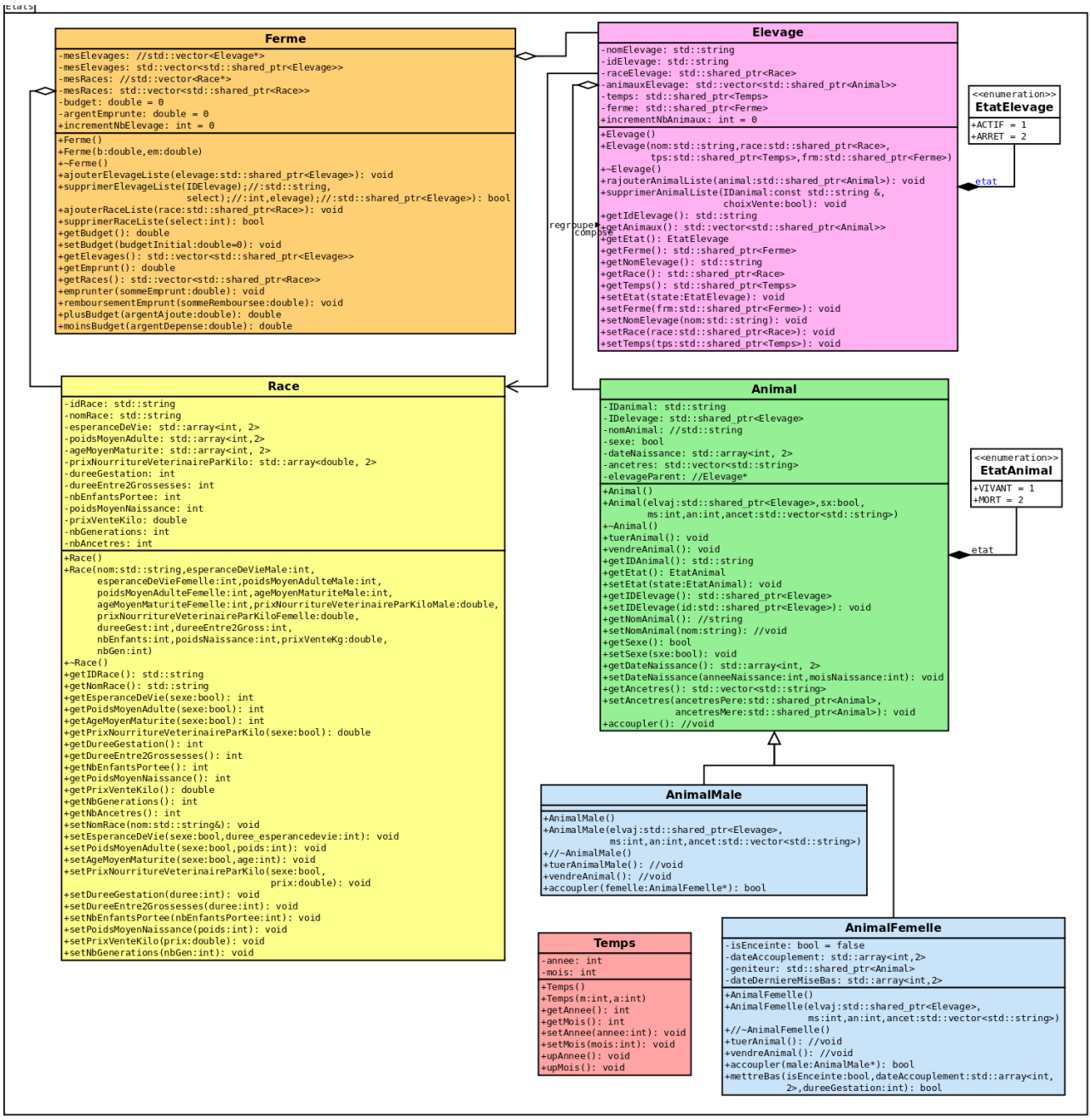
La classe **Race** (ou espèce) est celle qui représente les caractéristiques générales de tous les animaux de cette race, comme par exemple espérance de vie moyenne des mâles et des femelles de cette race, un poids moyen des bêtes adultes mâles/femelles, etc.
Une Race animale est donc caractérisée par des données qui lui sont propres.
Comme un élevage ne peut contenir qu'une race, il était question d'intégrer le contenu de la classe **Race** dans la classe **Elevage** , mais il doit être possible d'arrêter un élevage, puis de recommencer un élevage de cette même race sans avoir besoin de renseigner à nouveau toutes les données de cette race.

La classe **Animal** est celle qui définit chaque bête de la ferme par ses données personnelles : son âge, son poids, etc. Chaque bête appartient à un seul élevage. Les données regroupées dans la classe animal ne font pas de différences entre les sexes, ce sont les données que tous les animaux ont.
Chaque Animal est caractérisé par un état : Vivant ou Mort

Les animaux de sexes différents ont en plus des attributs et méthodes différentes : les classes **AnimalMale** et **AnimalFemelle** diffèrent par le fait qu'une femelle peut mettre au monde des petits.

Une classe **Temps**, gère la date, ce qui permet de faire évoluer les données de chaque animal.

1 Le logiciel n'étant pas un jeu vidéo, il n'y a pas d'états du jeu à proprement parler, comme ce qu'on peut trouver sur l'exemple donné sur Moodle


Etats
Ferme
-mesElevages: //std::vector<Elevage*>
-mesElevages: std::vector<std::shared_ptr<Elevage>>
-mesRaces: //std::vector<Race*>
-mesRaces: std::vector<std::shared_ptr<Race>>
-budget: double = 0
-argentEmprunte: double = 0
+incrementNbElevage: int = 0
+Ferme()
+Ferme(b:double,em:double)
+~Ferme()
+ajouterElevageListe(elevage:std::shared_ptr<Elevage>): void
+supprimerElevageListe(IDElevage);//:std::string,
select);//:int,elevage);//:std::shared_ptr<Elevage>): bool
+ajouterRaceListe(race:std::shared_ptr<Race>): void
+supprimerRaceListe(select:int): bool
+getBudget(): double
+setBudget(budgetInitial:double=0): void
+getElevages(): std::vector<std::shared_ptr<Elevage>>
+getEmprunt(): double
+getRaces(): std::vector<std::shared_ptr<Race>>
+emprunter(sommeEmprunt:double): void
+remboursementEmprunt(sommeRemboursee:double): void
+plusBudget(argentAjoute:double): double
+moinsBudget(argentDepense:double): double
Elevage
-nomElevage: std::string
-idElevage: std::string
-raceElevage: std::shared_ptr<Race>
-animauxElevage: std::vector<std::shared_ptr<Animal>>
-temps: std::shared_ptr<Temps>
-ferme: std::shared_ptr<Ferme>
+incrementNbAnimaux: int = 0
+Elevage()
+Elevage(nom:std::string,race:std::shared_ptr<Race>,
tps:std::shared_ptr<Temps>,frm:std::shared_ptr<Ferme>)
+~Elevage()
+rajouterAnimalListe(animal:std::shared_ptr<Animal>): void
+supprimerAnimalListe(IDanimal:const std::string &,
choixVente:bool): void
+getIdElevage(): std::string
+getAnimaux(): std::vector<std::shared_ptr<Animal>>
+getEtat(): EtatElevage
+getFerme(): std::shared_ptr<Ferme>
+getNomElevage(): std::string
+getRace(): std::shared_ptr<Race>
+getTemps(): std::shared_ptr<Temps>
+setEtat(state:EtatElevage): void
+setFerme(frm:std::shared_ptr<Ferme>): void
+setNomElevage(nom:std::string): void
+setRace(race:std::shared_ptr<Race>): void
+setTemps(tps:std::shared_ptr<Temps>): void
<<enumeration>>
EtatElevage
+ACTIF = 1
+ARRET = 2
etat
regroupe*
compose
Race
-idRace: std::string
-nomRace: std::string
-esperanceDeVie: std::array<int, 2>
-poidsMoyenAdulte: std::array<int,2>
-ageMoyenMaturite: std::array<int, 2>
-prixNourritureVeterinaireParKilo: std::array<double, 2>
-dureeGestation: int
-dureeEntre2Grossesses: int
-nbEnfantsPortee: int
-poidsMoyenNaissance: int
-prixVenteKilo: double
-nbGenerations: int
-nbAncetres: int
+Race()
+Race(nom:std::string,esperanceDeVieMale:int,
esperanceDeVieFemelle:int,poidsMoyenAdulteMale:int,
poidsMoyenAdulteFemelle:int,ageMoyenMaturiteMale:int,
ageMoyenMaturiteFemelle:int,prixNourritureVeterinaireParKiloMale:double,
prixNourritureVeterinaireParKiloFemelle:double,
dureeGest:int,dureeEntre2Gross:int,
nbEnfants:int,poidsNaissance:int,prixVenteKg:double,
nbGen:int)
+~Race()
+getIDRace(): std::string
+getNomRace(): std::string
+getEsperanceDeVie(sexe:bool): int
+getPoidsMoyenAdulte(sexe:bool): int
+getAgeMoyenMaturite(sexe:bool): int
+getPrixNourritureVeterinaireParKilo(sexe:bool): double
+getDureeGestation(): int
+getDureeEntre2Grossesses(): int
+getNbEnfantsPortee(): int
+getPoidsMoyenNaissance(): int
+getPrixVenteKilo(): double
+getNbGenerations(): int
+getNbAncetres(): int
+setNomRace(nom:std::string&): void
+setEsperanceDeVie(sexe:bool,duree_esperancedevie:int): void
+setPoidsMoyenAdulte(sexe:bool,poids:int): void
+setAgeMoyenMaturite(sexe:bool,age:int): void
+setPrixNourritureVeterinaireParKilo(sexe:bool,
prix:double): void
+setDureeGestation(duree:int): void
+setDureeEntre2Grossesses(duree:int): void
+setNbEnfantsPortee(nbEnfantsPortee:int): void
+setPoidsMoyenNaissance(poids:int): void
+setPrixVenteKilo(prix:double): void
+setNbGenerations(nbGen:int): void
Animal
-IDanimal: std::string
-IDelevage: std::shared_ptr<Elevage>
-nomAnimal: //std::string
-sexe: bool
-dateNaissance: std::array<int, 2>
-ancetres: std::vector<std::string>
-elevageParent: //Elevage*
+Animal()
+Animal(elvaj:std::shared_ptr<Elevage>,sx:bool,
ms:int,an:int,ancet:std::vector<std::string>)
+~Animal()
+tuerAnimal(): void
+vendreAnimal(): void
+getIDAnimal(): std::string
+getEtat(): EtatAnimal
+setEtat(state:EtatAnimal): void
+getIDElevage(): std::shared_ptr<Elevage>
+setIDElevage(id:std::shared_ptr<Elevage>): void
+getNomAnimal(): //string
+setNomAnimal(nom:string): //void
+getSexe(): bool
+setSexe(sxe:bool): void
+getDateNaissance(): std::array<int, 2>
+setDateNaissance(anneeNaissance:int,moisNaissance:int): void
+getAncetres(): std::vector<std::string>
+setAncetres(ancetresPere:std::shared_ptr<Animal>,
ancetresMere:std::shared_ptr<Animal>): void
+accoupler(): //void
<<enumeration>>
EtatAnimal
+VIVANT = 1
+MORT = 2
etat
AnimalMale
+AnimalMale()
+AnimalMale(elvaj:std::shared_ptr<Elevage>,
ms:int,an:int,ancet:std::vector<std::string>)
+//~AnimalMale()
+tuerAnimalMale(): //void
+vendreAnimal(): //void
+accoupler(femelle:AnimalFemelle*): bool
Temps
-annee: int
-mois: int
+Temps()
+Temps(m:int,a:int)
+getAnnee(): int
+getMois(): int
+setAnnee(annee:int): void
+setMois(mois:int): void
+upAnnee(): void
+upMois(): void
AnimalFemelle
-isEnceinte: bool = false
-dateAccouplement: std::array<int,2>
-geniteur: std::shared_ptr<Animal>
-dateDerniereMiseBas: std::array<int,2>
+AnimalFemelle()
+AnimalFemelle(elvaj:std::shared_ptr<Elevage>,
ms:int,an:int,ancet:std::vector<std::string>)
+//~AnimalFemelle()
+tuerAnimal(): //void
+vendreAnimal(): //void
+accoupler(male:AnimalMale*): bool
+mettreBas(isEnceinte:bool,dateAccouplement:std::array<int,
2>,dureeGestation:int): bool

III. Rendu du Logiciel

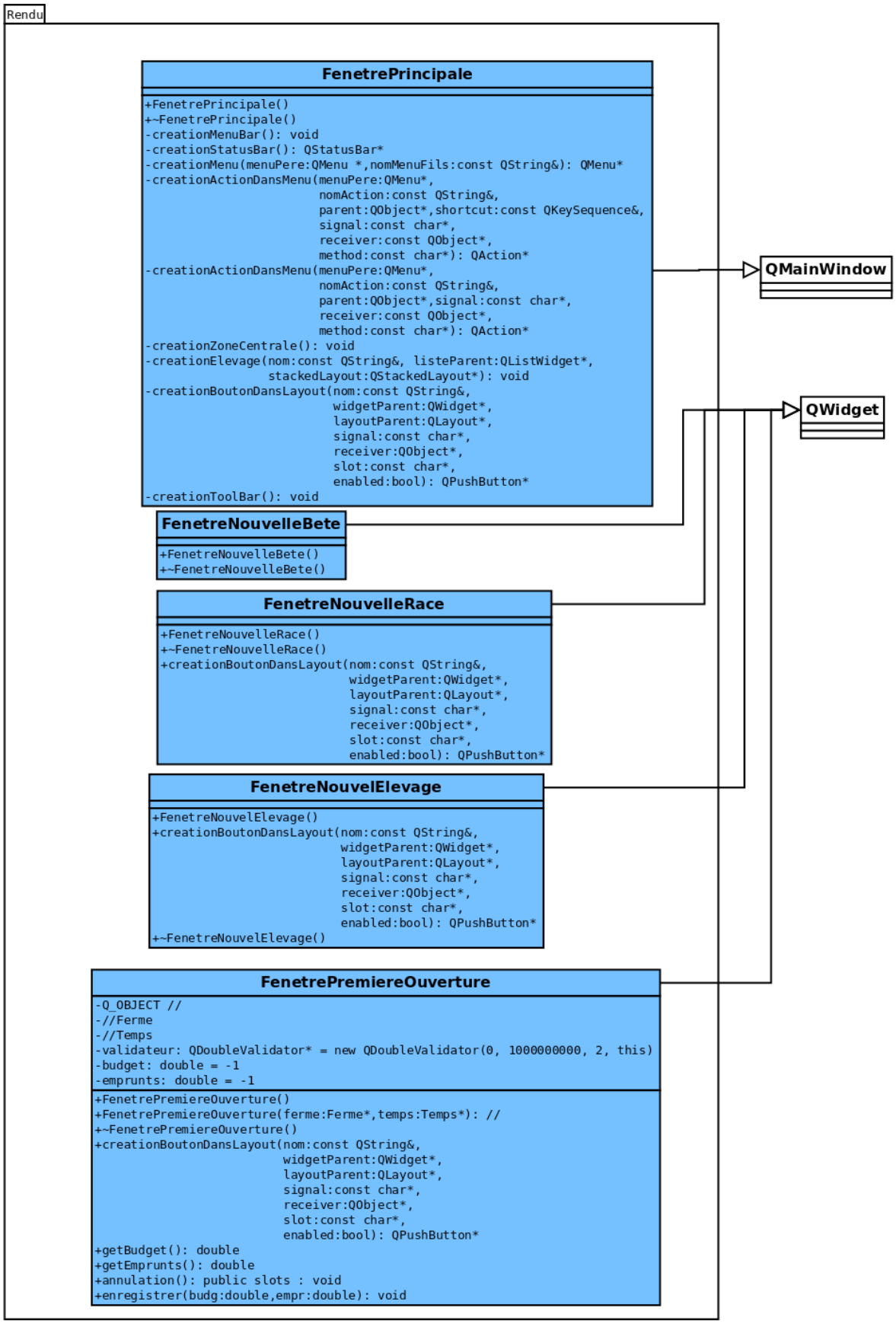
Le logiciel de gestion d'élevage, comporte une interface graphique dite « classique », ou interface « desktop ». Pour ce faire, j'utilise le framework Qt, qui convient parfaitement à ce genre d'application, et plus particulièrement la classe Qwidget, et nombreuses de ses classes filles.
Quand on lance le logiciel, une fenêtre principale s'ouvre. Elle contient une barre de Menu, une barre de Statut, une Toolbox. Au centre de la fenêtre (donc ce qui est la véritable raison du logiciel) se trouvent la liste des élevages contenus dans la ferme (à gauche de l'écran), et la description graphique de l'élevage (à droite de la liste), qui apparaît sous forme de tableau.
Quand on créée un nouvel élevage, une nouvelle fenêtre apparaît, et permet de sélectionner des caractéristiques de l'élevage (comme par exemple la race animale qui sera élevée dans cet élevage). Il faut ensuite directement ajouter des animaux à cet élevage (car un élevage sans animal ne peut exister). Quand on veut créer un nouvel animal à cet élevage, une nouvelle fenêtre apparaît. C'est une fenêtre de création d'animal dans laquelle on renseigne des informations sur l'animal.
Si, quand on créée un élevage, la race qu'on élève est inconnue, une nouvelle fenêtre apparaît, dans laquelle on enregistre toutes les caractéristiques de cette race, desquelles dépendront tous les animaux de l'élevage.

Conception Logiciel

Fenêtre principale : Le cœur du rendu réside dans cette classe, car elle créée la fenêtre principale du logiciel. Elle hérite des propriétés de la classe QMainWindow proposée par le framework Qt. Son unique méthode est son constructeur, qui construit aussi les éléments graphiques de la fenêtre.

Les classes **NouvelleFenetreBete**, **NouvelleFenetreRace**, **NouvelleFenetreElevage**, ont pour rôle de créer les fenêtres qui servent à créer les nouveaux élevages, les nouvelles races élevées, et les nouvelles bêtes dans les élevages. Ces classes ne contiennent que leurs constructeurs respectifs, qui leur permet de se créér, ainsi que les Widgets qu'elles contiennent.

Diagramme des classes de rendu



IV. Règles de changement d’états, et moteur de jeu.

Changements extérieurs

Les changements extérieurs sont provoqués par l’utilisateur du logiciel : créer un nouvel élevage, supprimer des animaux, rajouter de nouveaux animaux, les faire s’accoupler, rajouter des races à la base de données de races. Des commandes sur le rendu sont aussi disponibles, comme afficher/masquer des fenêtres.Telles sont les commandes auxquelles a accès l’utilisateur, via les boutons de l’interface graphique.

Changements autonomes

Dans le logiciel, l’unité de temps est le mois. Dans un soucis de simplicité, l’utilisateur fait avancer le temps mois par mois, à l’aide d’une touche prévue à cet effet. Cela laisse donc le temps à l’utilisateur d’effectuer toutes les manipulations qu’il désire sur ses élevages. Une fois ces manipulations effectuées, il passe au mois suivant, et les états changent automatiquement. Par exemple si une femelle était enceinte et qu’elle met bas quand le terme de la grossesse arrive, les nouveaux animaux sont rajoutés automatiquement à la base de donnée. De même qu’un animal trop vieux meurt quand il atteint la limite d’âge imposée par l’espérance de vie de l’espèce à laquelle il appartient. Chaque mois, sont soustraits au budget général de la ferme, les frais de nourriture et de vétérinaire nécessaires pour chaque animal (ce total est évidemment calculé en fonction de son poids)

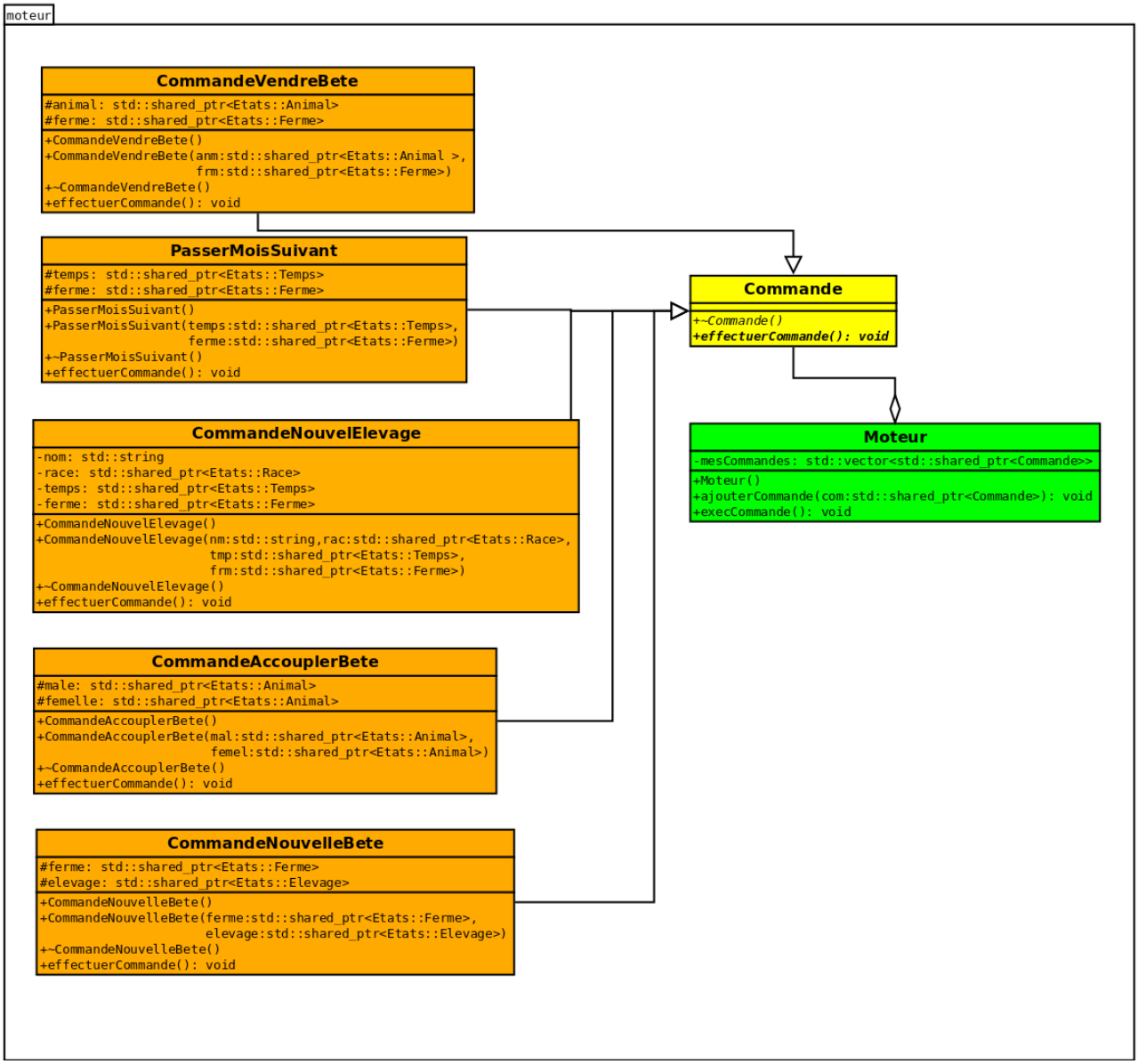
Conception logiciel

.
L'ensemble du moteur de jeu repose sur un patron de conception de type Commande, et a pour but la mise en œuvre différée de commandes extérieures sur l'état du jeu.

Classes **Commande<Nom de la commande>**.
Le rôle de ces classes est de représenter une commande extérieure, provenant de l’interface graphique : ce sont les différents boutons de cette interface graphique qui fabriquerons les instances de ces classes.

Engine.
C'est le coeur du moteur. Elle stocke les commandes dans un std::vector, et quand est appelée la méthode *execCommande()*, le vector des commandes est vidé, et est appliquée à chaque commande qui sort la méthode *effectuerCommande()*.
La commande PasserMoisSuivant créée les commandes pour les changements autonomes du jeu, et rappelle ensuite la méthode *execCommande()* du moteur.

Diagramme des classes du moteur



V. Intelligence artificielle

Stratégie

L'intelligence artificielle du logiciel, doit prendre en compte le budget de la ferme, les différents élevages, et les animaux de chaque élevage. Avec ces animaux, elle peut décider de les vendre, de les faire s'accoupler, ou d'acheter de nouvelles betes (pour renouveler la diversité génétique dans les élevages, par exemple).

Une IA simple peut effectuer des commandes aléatoires, avec une obligation d'envoyer la commande **PasserMoisSuivant**, toutes les *n* actions (*n* dépend du nombre total d'animaux dans la ferme). Une telle IA ne peut évidemment pas faire tenir une ferme longtemps, très rapidement il n'y aurait plus aucun animal dans la ferme (ou alors on arriverait au cas où il ne reste plus que des animaux d'un seul sexe dans certains élevages, et si l'IA n'a pas la présence d'esprit d'acheter un animal du sexe manquant, toutes les bêtes finiraient par mourir), ou même la ferme se retrouverait sans budget, si les actions choisies aléatoirement consistent en l'achat de nouveaux animaux.

Conception Logiciel

VI. Parallélisation

Répartition sur plusieurs threads

Dans le projet, tout se passe à la base sur un unique thread. Le but est de répartir le travail sur plusieurs threads.

J’ai donc créé deux threads sur mes fonctions qui actualisent le rendu, et qui exécutent mes commandes

Commandes. Ce sont les boutons de l’interface Qt. Chaque bouton, quand il est cliqué, créée un objet Commande, qui est ajouté à un vector de commandes. Mon thread sur les commandes est en réalité une boucle infinie qui vide le vector de commande grâce à l’opération push_back(), et qui effectue l’action affiliée à la commande, avant de la vider du vector.

Rendu. Quand une commande est vidée du vector, elle créée une notification de changement d’état. Cette notification est stockée dans un vector de notifications. Mon thread pour le rendu est une boucle infinie qui vide en permanence mon vector de notifications, et qui actualise le rendu en fonction de la notification : si la notification indique que tout a changé, toutes les betes de tous les élevages sont rajoutée aux layout Qt de chaque élevage. La liste des élevages présente dans l’interface est aussi mise à jour, ainsi que le budget de la ferme, la date, etc.. Si la notification indique que la liste des élevages a changé, ou juste que des animaux ont changé, respectivement la liste des élevages, et la liste des animaux (de l’élevage dont les animaux ont changé d’état), sont actualisés

VI. API Web

Il nous faut créer une architecture client /serveur.

Tous les clients ont sur leurs machines une IHM, un état de logiciel, et un moteur de logiciel.

Quand le client créée une commande, elle est normalement envoyée dans le moteur de jeu sur la machine, sauf qu'il faut désormais envoyer la commande (grâce à une API REST) au serveur.

La commande (si elle est acceptée par le serveur) est alors stockée dans une liste dans le serveur. Chaque commande est accompagnée d'un identifiant client.

Le serveur peut refuser des commandes si elles sont contraires à des commandes déjà envoyées par d'autres utilisateurs.

Le client envoie régulièrement (automatiquement) des requêtes afin de récupérer les dernières commandes envoyées par les autres utilisateurs (et conservées dans l'historique), afin de mettre à jour les états du logiciel se trouvant sur sa machine, et par conséquent pour actualiser le rendu.

À chaque utilisateur ne sont envoyées que les commandes qui ont été stockées depuis la dernière fois qu'il a actualisé les états sur sa machine.

Le serveur contient un moteur de logiciel et les états du logiciel (les mêmes que ceux qui se trouvent sur chaque machine client.

Seules les commandes sont transmises entre client et serveur, pas de notification de changement de rendu.

Les changements d'état automatiques, ne sont pas transmis, chaque client a un moteur du logiciel sur sa machine. Chaque moteur effectue donc ces changements seul.

Service REST qui demande un identifiant d'utilisateur en début d'utilisation, et qui réceptionne l'identifiant et l'état actuel de la ferme

Requête

Méthode GET / idUser

Donnée : Pas de données

Réponse

- Cas où personne n'a ouvert le logiciel avant

  Status : 404 NOT FOUND
  Donnée :

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type":"number",
    "minimum":1,
    "maximum":5
}
```

- Cas où un autre utilisateur manie aussi l'élevage :

  À l'ouverture du logiciel, on suppose que le logiciel a déjà été ouvert sur un autre ordinateur, et que les listes d'élevages sont déjà remplies. En effet on peut supposer que dans notre ferme d'élevages, le directeur a ouvert le logiciel dans son bureau pour constater l'état actuel de l'élevage, et qu'un employé l'a aussi ouvert pour gérer l'alimentation des animaux. Quand le programme se lance, on ne suppose pas qu'on lance une partie au début avec un autre utilisateur aussi connecté a serveur avec le programme

  Status 200 OK

  Donnée :

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type":"number",
    "minimum":1,
    "maximum":5
}
```

- Cas où plus de 5 utilisateurs sont connectés sur le gestionnaire d'élevages

  Status 403 FORBIDDEN

  Donnée : Pas de données.

Service REST qui envoie les commandes et qui réceptionne les statuts des commandes

Requêtes

Méthode PUT/commande

Données envoyées : On envoie la commande au moteur.

```
{
	"$schema": "http://json-schema.org/draft-04/schema#",
	"type": "object",
	"properties":
	{
		"commande":
		{
			"type": "object",
			"properties":
			{
				"nomcommande":{"type":"string"},
				"ferme":{"type":"boolean"},
				"nomElevage":{"type":"string"},
				"race":{"type":"string"},
				"animal":{"type":"string"},
				"animal":{"type":"string"},
				"budget":{"type":"number"},
				"emprunts":{"type":"number"},
				"sexe":{"type":"boolean"},
				"mois":{"type":"number", "minimum":0, "maximum":11},
				"annee":{"type":"number", "minimum":2016, "maximum":2500},
				"ancetre":{"type":"array", "items":{"type":"string"}},
				"fenetreprincipale":{"type":"boolean"},
				"observateur":{"type":"boolean"}
			}
		},
		"idUser":
		{
			"type":"number",
			"minimum":1,
			"maximum":5
		}
	},
	"required": ["commande", "idUser"]
}
```

Réponses

- Cas où la commande est autorisée

  Status 200 OK

  Données : le serveur renvoie l'index de la commande dans l'historique. (Dans une évolution ultérieure du logiciel, identifier chaque commande individuellement s'avère impossible car le logiciel doit pouvoir conserver la base de données, sur plusieurs mois.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type":"number",
    "minimum":1,
    "maximum":5
}
```

- Cas où la commande n'est pas autorisée parce qu'en contradiction avec une commande antérieure

  Status 409 CONFLICT

  Donnée : Pas de donnée

- Cas où le logiciel n'a pas été initialisé, toutes les commandes sont rejetées, sauf la commande d'initialisation

  Status 503 SERVICE UNAVAILABLE

  Donnée : Pas de donnée

Service REST qui demande au serveur quelles commandes ont été réceptionnée par le serveur et qui avaient été envoyées par les utilisateurs, réceptionne les commandes reçues par le serveur

Requêtes

Méthode : GET / commandes_serveur / <identifiant : number , minimum : 1, maximum : 5>

Données : Pas de données

Réponses

- Cas où l'identifiant est invalide :

  Status 400 BAD REQUEST

  Donnée : Pas de données.

- Cas où l'identifiant est valide et si il n'y a aucune nouvelle commande n'a été stockée depuis la dernière requête similaire

  Status 204 NO CONTENT)

  Donnée : Pas de données.

- Cas où l'identifiant est valide, et si des commandes ont été rajoutées depuis la dernière mise à jour des états du logiciel.

  Status 200 OK

  Données : On reçoit, un tableau contenant les dernières commandes stockée dans le moteurs depuis la dernière requête du même type.

```
{
        "$schema": "http://json-schema.org/draft-04/schema#",
        "type": "object",
        "properties":
        {
                "mescommandes":
```

```
                {
                        "type":"array",
                        "minItems":1,
                        "items":
                        {
                                "type":"object",
                                "properties":
                                {
                                        "commande":
                                        {
                                                "type": "object",
                                                "properties":
                                                {
                                                        "nomcommande":{"type":"string"},
                                                        "ferme":{"type":"boolean"},
                                                        "nomElevage":{"type":"string"},
                                                        "race":{"type":"string"},
                                                        "animal":{"type":"string"},
                                                        "animal":{"type":"string"},
                                                        "budget":{"type":"number"},
                                                        "emprunts":{"type":"number"},
                                                        "sexe":{"type":"boolean"},
                                                "mois":{"type":"number", "minimum":0, "maximum":11},
                                        "annee":{"type":"number", "minimum":2016, "maximum":2500},
                                                "ancetre":{"type":"array", "items":{"type":"string"}},
                                                        "fenetreprincipale":{"type":"boolean"},
                                                         "observateur":{"type":"boolean"}
                                                }
                                        },
                                        "idUser":
                                        {
                                                "type":"number",
                                                "minimum":1,
                                                "maximum":5
                                        }
                                },
                                "required": ["commande", "idUser"],
                        }
                }
        },"required":["mescommandes"],
}
```