RAPPORT DE PROJET

Atlas entomologiste régional

WIBAUX Guillaume	FRANCOIS Fabien
BRANDT Nicolas	VEDEAU Guillaume
RAYNAUD DE FITTE David	HAMADI Merwane
ZEJLY El mehdi	

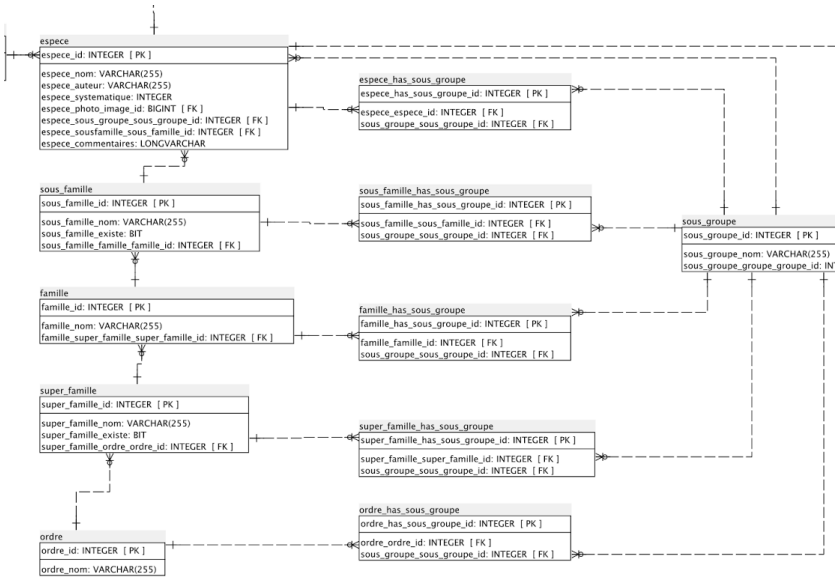
Sommaire

1. Contexte et définition du problème

L'association des entomologistes régionale recense toutes les observations d'insectes dans sa zone depuis des décennies, dans le but d'éditer son atlas. Ceci se fait via une base de données dans laquelle les membres enregistrent leurs témoignages et en renseignent les détails. Cette base est maintenant utilisée par une application développée sous le framework PLAY par des élèves de l'école Centrale.

Cependant, les difficultés à définir la base de données ont entraîné certains problèmes logiques dans sa construction, notamment au niveau des hiérarchies utilisées par les entomologistes. En effet deux classifications sont utilisées, une locale, propre à l'association nantaise, et l'autre, scientifique, commune à tous les entomologistes. Dans la base de données de départ, chaque élément/noeud de chacune des classifications est contenu dans une table différente selon sa "profondeur" dans l'arbre.

L'extrait ci dessous présente la partie de la base de données initiale du projet concernant l'organisation des espèces :



Pourtant, après réflexion, il faut comprendre que se ne sont que des "tiroirs" dans lesquels on range les différentes espèces observées ; et chaque niveau de tiroir n'est différent d'un autre que par celui dans lequel il est inclus (celui ou ceux qu'il comprend, au niveau n-1, sont eux même défini par lui même au niveau n), et ce

jusqu'aux espèces. Tous les niveaux d'une même organisation doivent ainsi pouvoir être construits et utilisés de la même manière.

De plus l'organisation actuelle de la base entraîne des complications dans l'utilisation de l'application. Un exemple frappant est l'utilisation des sous-familles. Dans la configuration initiale, une espèce doit appartenir à une sous-famille, bien que cela ne soit pas obligatoire dans la réalité. La difficulté a été contournée par des sous familles "fictives", passant ainsi le nombre de vingt sous-familles réellement existantes à plus de cent sous-familles utilisées. On comprend bien que ceci rend l'application plus complexe d'utilisation et surtout moins intuitive.

L'application initiale, bien que fonctionnelle, est donc basée sur une base un peu bancale, et ceci a des conséquences sur les possibilités qu'elle offre. Modifier ou ajouter des fonctionnalités devient plus complexe car il faut nécessairement travailler sur un grand nombre de tables. Sa base de données en partie inadaptée rend l'application très peu évolutive ; or celle ci est destinée à être davantage développée dans le futur.

Mais modifier ces organisations représente un travail conséquent, car, à cause d'une forme d'effet papillon, modifier la base de données entraîne de nombreuses modifications de l'application. Une majorité de ces fonctionnalités prend en compte la base initiale, et il faut prendre en compte chacune de ces modifications.

Enfin certaines fonctionnalités, entièrement basée sur cette organisation, sont entièrement à reprendre puisque bâties sur cette même erreur de logique.

2. Objectifs

Nous avons classifié les tâches spécifiées au préalable dans le cahier des charges.

Cette priorisation prend particulièrement d'importance dans ce type de projet, où l'on ne peut pas toujours avancer à plusieurs de front, et où certaines modifications peuvent impacter de manière considérable d'autre tâches.

Priorité	Tâche
1	● Modification du modèle conceptuel de la base de données ● Développement du script de migration ● Modification du programme en conséquence
2	● Permettre l'édition d'informations (villes etc…) ● Permettre l'identification des fiches papiers
3	● Développer les exports (excels)

A partir de ceci nous avons pu établir un planning prévisionnel, basé sur la segmentation des tâches précédente. Il a fallut de plus tenir compte du temps nécessaire à la prise en main du framework PLAY! Et à la compréhension de l'application dans son état initial sachant que la documentation est limitée.

Bien que globalement respecté, ce planning a néanmoins été sujet à plusieurs ajustements mineurs.

3. Travail Effectué

3.1. Modification de la base de données

La base de données, sur laquelle est construite notre application, doit être modifiée en premier. Il est nécessaire de créer un nouveau modèle, plus logique. Le but de cette modification est de n'utiliser qu'un seul élément pour stocker les informations que l'on souhaite étudier. Seul un attribut de "type" permettra de distinguer le niveau hiérarchique de ce qu'il contient, à la manière d'une étiquette sur une boite.
Une fois la nouvelle base réalisée, un script de migration devra être mis au point, de manière à transférer les données de l'ancienne vers la nouvelle base, en les "rangeant" correctement et sans perte d'information.

3.1.1. Définition d'un nouveau schéma

Dans un premier temps, nous avons réfléchi à une nouvelle architecture pour la base de données. Plusieurs solutions ont été proposées. Voici les idées principales de la solution retenue (proposition n°2) :

- Regrouper les différents groupements (scientifiques ou locaux), dans une seule table (une pour les scientifiques et une pour les locaux)
- Le type de groupement pour ces tables sera défini par une relation avec une table type_groupement_scientifique ou type_groupement_local, via une clé étrangère.
- Enfin, une clé étrangère sera intégrée dans cette table pour indiquer le père de chaque élément. Le fait d'avoir tous les groupements dans la même table permet d'être plus souple dans la hiérarchie des groupements : passer d'espèce à famille ne nécessite plus de créer une sous-famille intermédiaire fictive.

Ces modifications permettent d'apporter à la base de données un caractère plus évolutif.

L'éventualité de regrouper les deux hiérarchies de groupements a été soulevée (proposition n°3), mais non retenue, pour conserver un modèle plus intuitif aux yeux des non informaticiens.

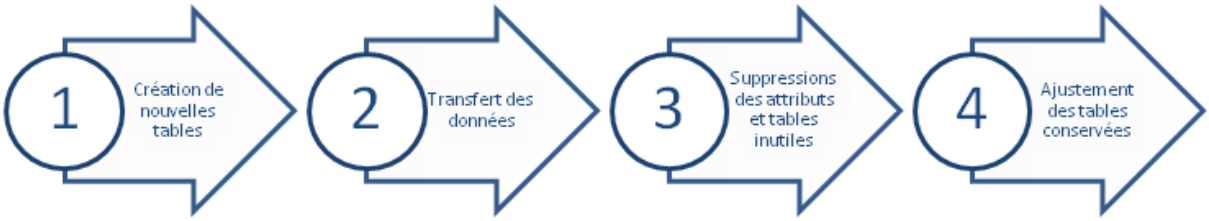
Les schémas complets des bases de données (l'ancienne, la nouvelle et les propositions étudiées) se trouvent en annexe.

3.1.2. Migration

3.1.2.1. Tâches à réaliser

Maintenant que l'architecture de la nouvelle base est définie, il faut définir comment l'atteindre, et quels outils seront nécessaires pour y arriver. Voici l'enchaînement d'actions que nous avons choisi pour le script :

Si la création des nouvelles tables ne pose pas de soucis particuliers, il



n'en va pas de même pour le transfert des données. L'architecture choisie fait que les IDs des groupements vont changer par rapport à l'ancienne base. Or, ces IDs vont être nécessaires pour l'insertion dans la base des groupements fils. Il faut donc prévoir un système permettant de récupérer les nouveaux IDs, sachant que l'accès aux anciens IDs ne sera pas un problème, puisqu'ils sont utilisés dans les clés étrangères des tables dont nous voulons transférer les données. Pour palier à cette difficulté, nous avons pensé à utiliser une table de hachage, qui permettra d'avoir le nouvel ID d'un objet, à partir du moment où nous avons l'ancien.
Nous allons utiliser un autre outil dans le script de migration : les boucles conditionnelles. En effet, cela sera nécessaire pour traiter le problème des sous-familles et super-familles fictives présentes dans l'ancienne base.

3.1.2.2. Principe du script

Le script de migration que nous avons développé est un programme Java, créé sur l'IDE Netbeans. Le choix de Java pour implémenter le script est discutable, notamment car le côté programmation orientée objet n'est absolument pas utilisé. Ce choix a été motivé notamment par l'habitude que nous avons de travailler avec Java. De plus, un script SQL pur nous semblait compliqué à mettre en oeuvre, du fait des outils dont nous avions besoin dans le script, et Java fournissait tous ces outils : boucles, connexions à une base de données et tables de hachage.3.1.2.4 Le principe du programme Java est simple : la méthode main initialise la connexion à la base de données, puis appelle les autres fichiers qui créent et envoient les différentes requêtes à la base de données. La méthode main est également celle qui récupère les exceptions SQL en cas de problème avec la base de données. Toutes les méthodes créant les requêtes, ainsi que les tables de hachage, sont déclarées en static, pour éviter d'avoir à créer des instances des différentes classes, ce dont nous n'avons pas besoin.

3.1.2.3. Modifications du script

L'objectif de cette partie est de détailler les modifications que nous avons effectuées sur le script, qui ont conduit à la version finale dont l'algorithme est disponible en annexe.

L'algorithme du script a été modifié plusieurs fois, pour diverses raisons :

- La principale modification a été la réutilisation des anciennes tables, en les renommant simplement au lieu de créer des nouvelles tables, dans le but d'éviter des transferts de données inutiles.
- Une autre modification importante a été de garder certains noms tels qu'ils étaient avant, pour limiter le nombre de changements de nom à effectuer dans l'application.
- Il y a eu quelques autres modifications mineures, comme le retrait de la contrainte NOT NULL sur l'attribut utm de la table fiche.

3.1.2.4. Tests effectués

Nous avons réalisé les tests du script de migration sur des bases de données locales créées à l'aide de Xampplite, de MySQLWorkbench, et du fichier Backup.sql de la base AER qui nous avait été fourni par Jean-Yves MARTIN. Nous avons ensuite vérifié que le script avait bien le comportement escompté, grâce à des requêtes SQL, et en comparant les données de la nouvelle base avec celles de l'ancienne.

Une fois satisfaits par les résultats que nous obtenions avec le script de migration, nous avons testé les fonctionnalités de l'application sur la nouvelle base de données. C'est à ce moment qu'est apparu le problème de synchronisation entre les noms de la base de données et les noms de l'application. Le projet étant limité dans le temps, nous avons décidé de garder certains noms de l'ancienne base de données, dans le but de gagner du temps, ce qui nous a permis d'avancer plus dans les objectifs du projet.

3.2. Fonctionnalités de l'application

3.2.1. Modifications liées à la modification de la base

Après une analyse de l'application, nous avons remarqué que :

- Les modifications liées à la classification scientifique (famille, sous-famille ...) n'ont qu'un impact réduit sur l'application (relativement peu de méthodes sont à modifier)
- En revanche, les modifications liées à la classification locale (groupe et sous-groupe) impactent la quasi totalité de l'application.

3.2.2. Modification des fonctionnalités existantes

Un administrateur a entre autres accès à une interface de gestion des classifications, que ce soit locale ou scientifique. Cependant, à cause des difficultés de définition de la base de données rencontrés par les précédents développeurs de l'application, nous avons du modifier cette fonctionnalité.

En effet, elle a été construite en organisant les différents éléments des classifications comme étant des “objets” distincts, nécessitant ainsi ses fonctions propres.

Notre nouvelle base de données est ainsi faite que ces “blocs” de chacune des deux organisations soient identiques : ils ne sont distingués que par leurs “type_groupement”. Le code nécessaire à la manipulation de ces blocs en est ainsi simplifié : on doit dorénavant implémenter le code responsable de la manipulation d’un bloc, sans tenir compte de sa place dans la classification à laquelle il appartient.

Ceci nous a conduit, une fois la nouvelle base crée, à reconstruire complètement les vues et les controllers associés aux modèles des classifications. Nous avons donc profité de cette opportunité pour rendre l’espace de gestion des groupements locaux et scientifiques plus intuitif. De plus, il a fallu reprendre de zéro les méthodes pour l’ajout, la modification et la suppression afin de prendre en compte les nouveaux attributs et les nouvelles relations du groupe dans la base de données.

4. Gestion de projet

4.1. Difficultés

Nous avons rencontré des difficultés nouvelles : outre les problèmes techniques, l'organisation et le timing ont joué des rôles primordiaux dans le déroulement du projet.

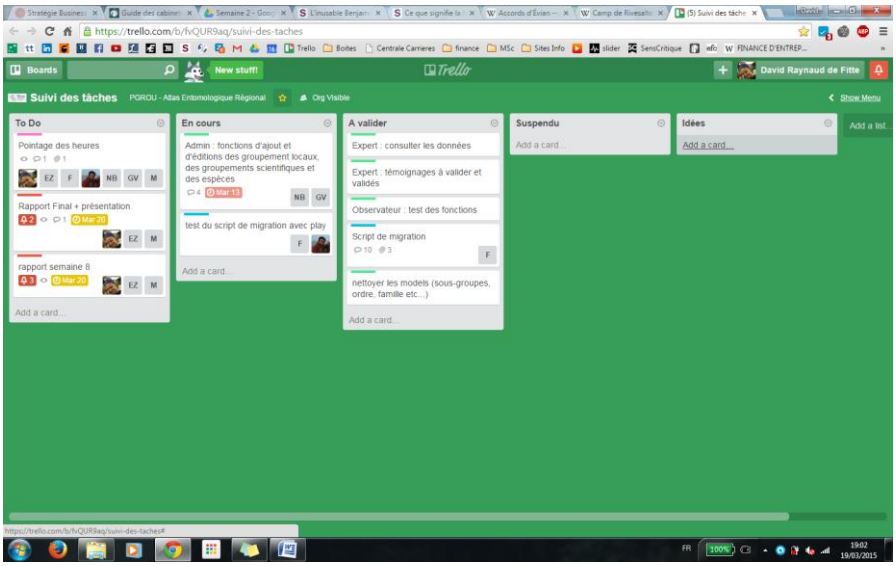
En effet, certains membres du groupe travaillant déjà sur cette application, il a fallut prendre du temps afin que tous soient au même stade de compréhension de la problématique.

De plus, la maîtrise du framework Play! fut un enjeux majeur du problème : il a en effet fallut que les éléments du groupe dont le travail portait sur l'application elle même prennent le temps de se familiariser avec cet outil.

4.2. Trello & GitHub

Travailler à sept pose également le problème de la communication. Nous nous sommes donc organisés en utilisant un outil de gestion.

Nous avons choisi Trello, application intuitive qui prend la forme d'un tableau d'organisation et de synthèse, qui nous a permis d'échanger sur toutes nos tâches en cours et d'avoir une vision d'ensemble de notre avancement. Mais un bon croquis vaut mieux qu'un long discours :



De même, pour partager le code, nous nous sommes servis du service web utilisé en cours, GitHub.

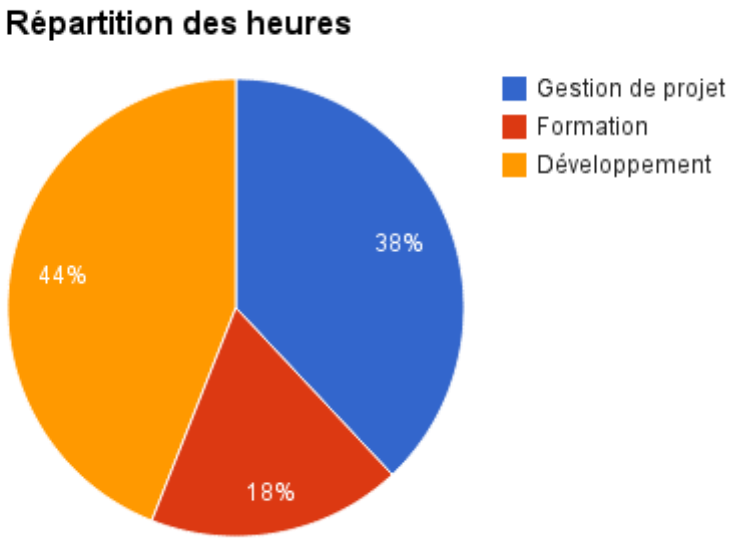
4.3. Suivi des heures

Parallèlement, nous avons mis en place un dispositif de suivi des heures permettant de dissocier par personne et par semaine, différentes catégories de travail :

- Rapports, management de projet, réunions,
- Conception, tutoriels (non productif)
- Développement (programmation, débogage)

Ceci a permis de mettre en valeur l'importance prise par la gestion de projet qui correspond à environ 40% du temps passé par l'ensemble des élèves sur le projet.

Le graphique ci dessous rapporte le pourcentage d'heures passées par catégories :

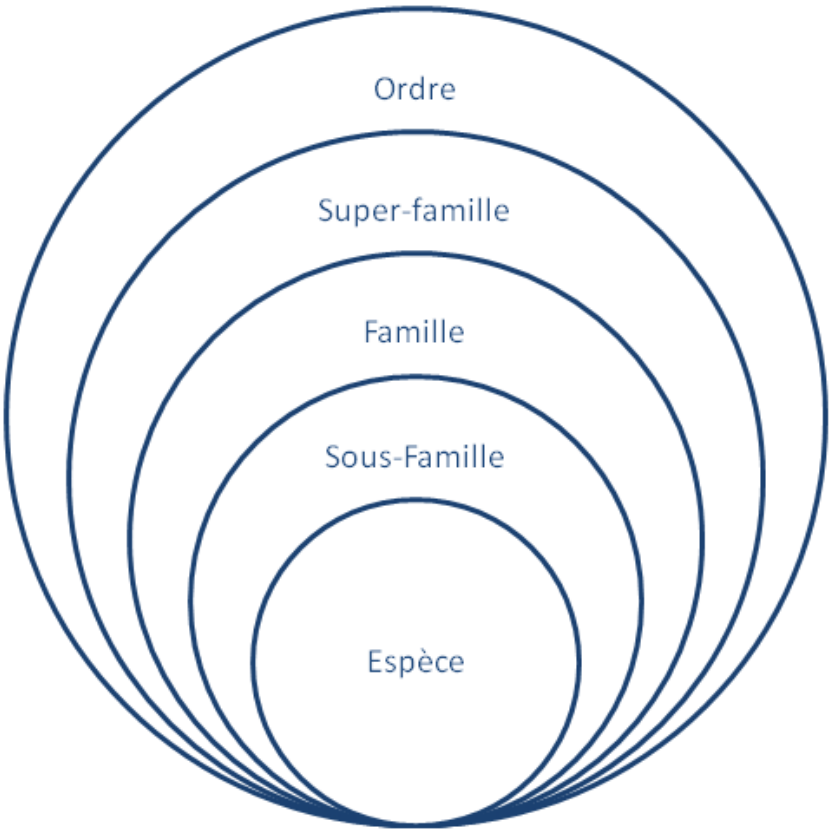


5. Annexes

5.1. Hiérarchisation et organisation des espèces

Les espèces observées sont organisées de deux manières distinctes :

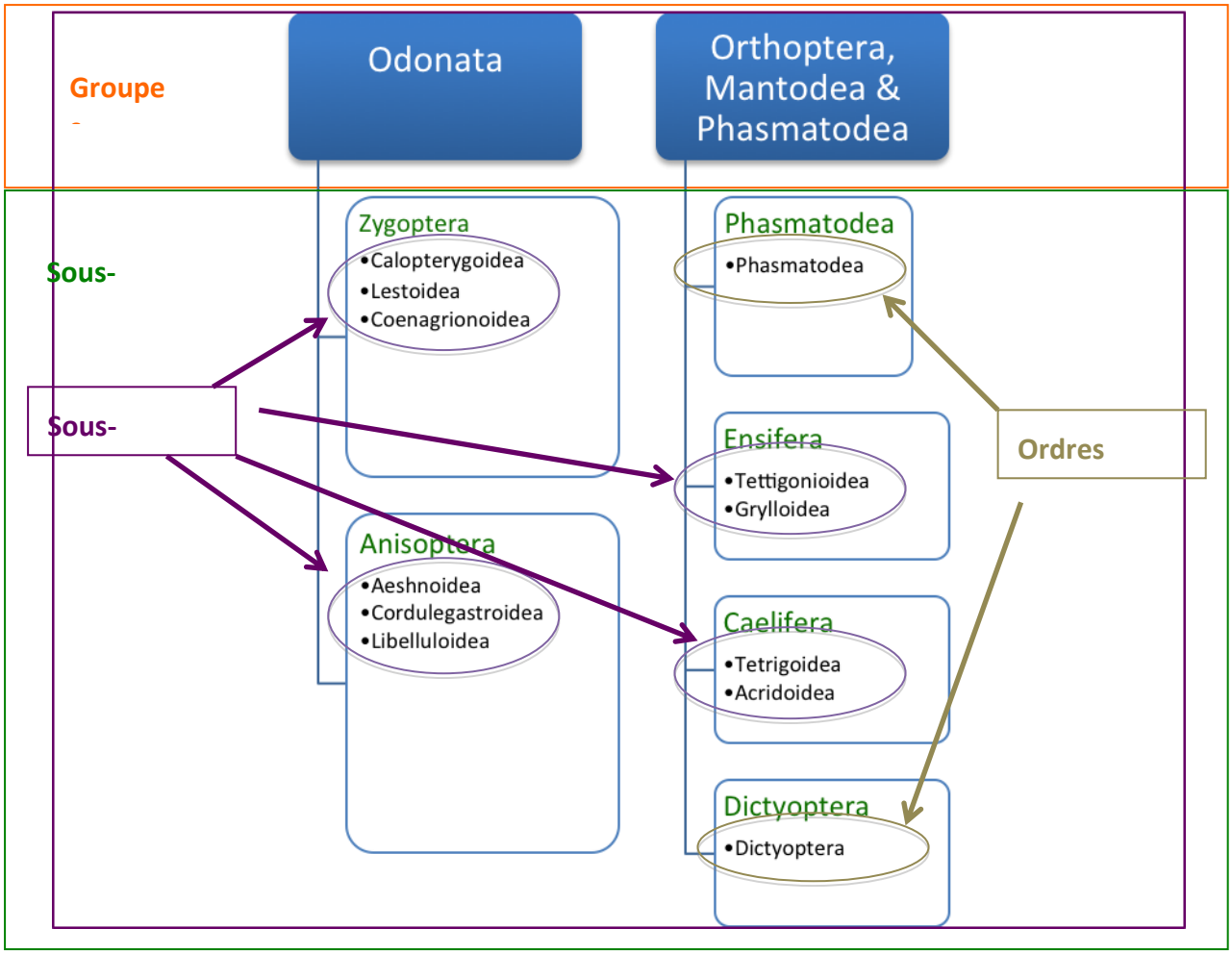
- selon la classification scientifique des espèces
- selon une organisation en « groupes » et « sous-groupes ».



Classification scientifique

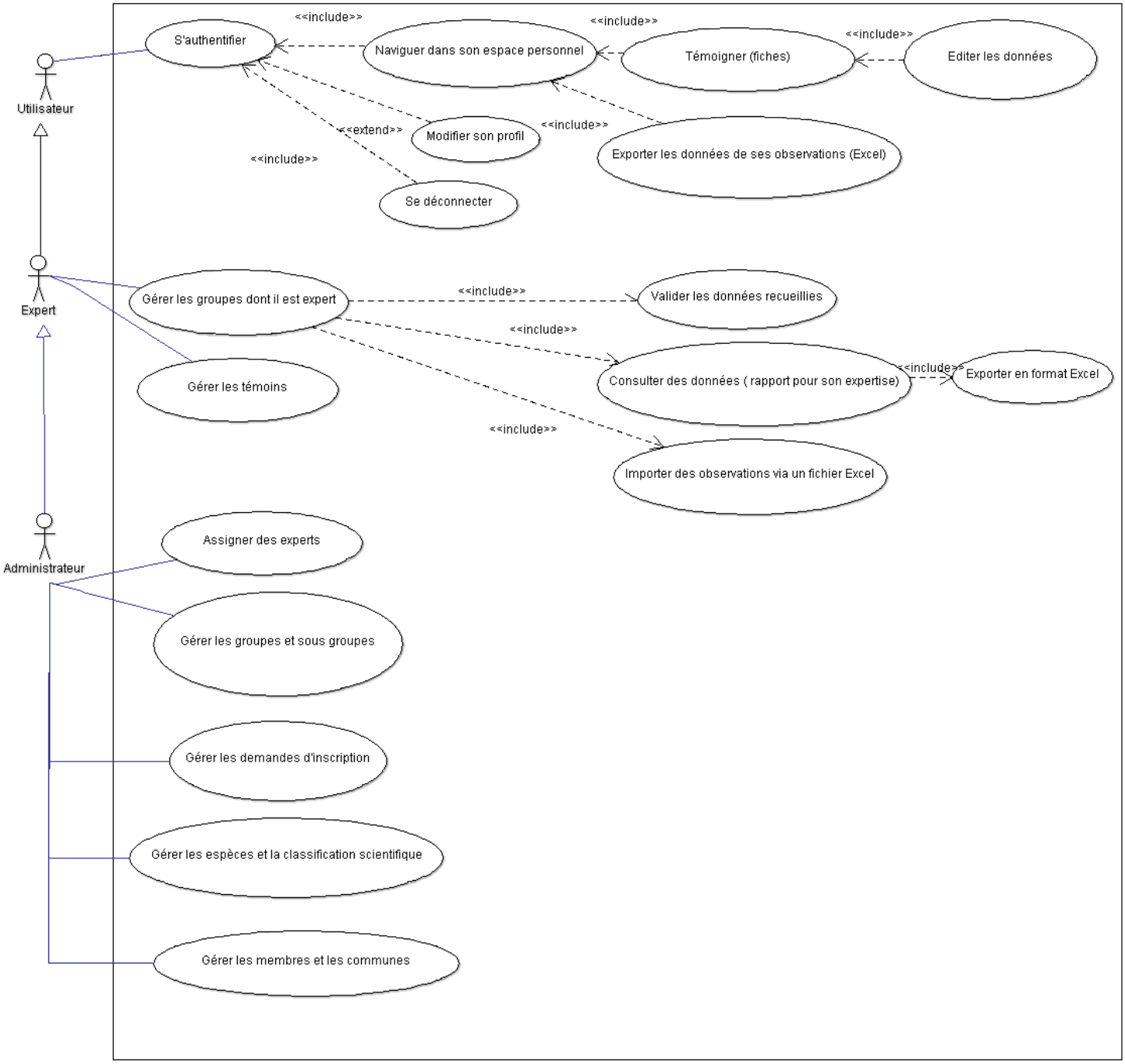
L'organisation par « groupes» est complémentaire de la classification scientifique et est spécifique à l'association. Elle permet de regrouper sous un élément de même niveau

hiérarchique (les sous-groupes) des éléments de niveaux différents dans la classification scientifique (espèces, familles, ordres…).



Exemple de groupes et sous-groupes

5.2. Diagramme des cas d'utilisation de l'application



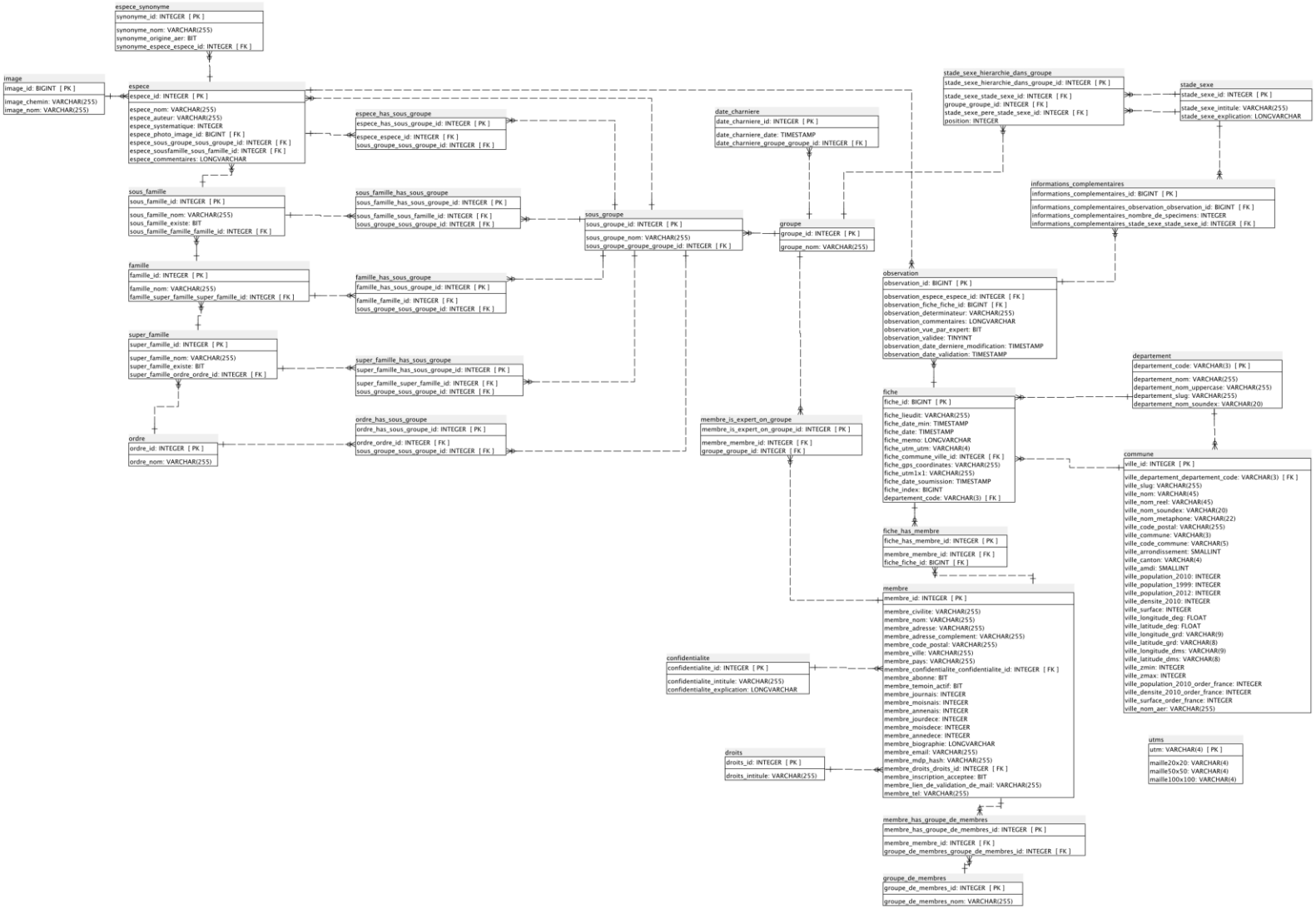
5.3. Scénario de modifications des observations

Processus de validation des observations

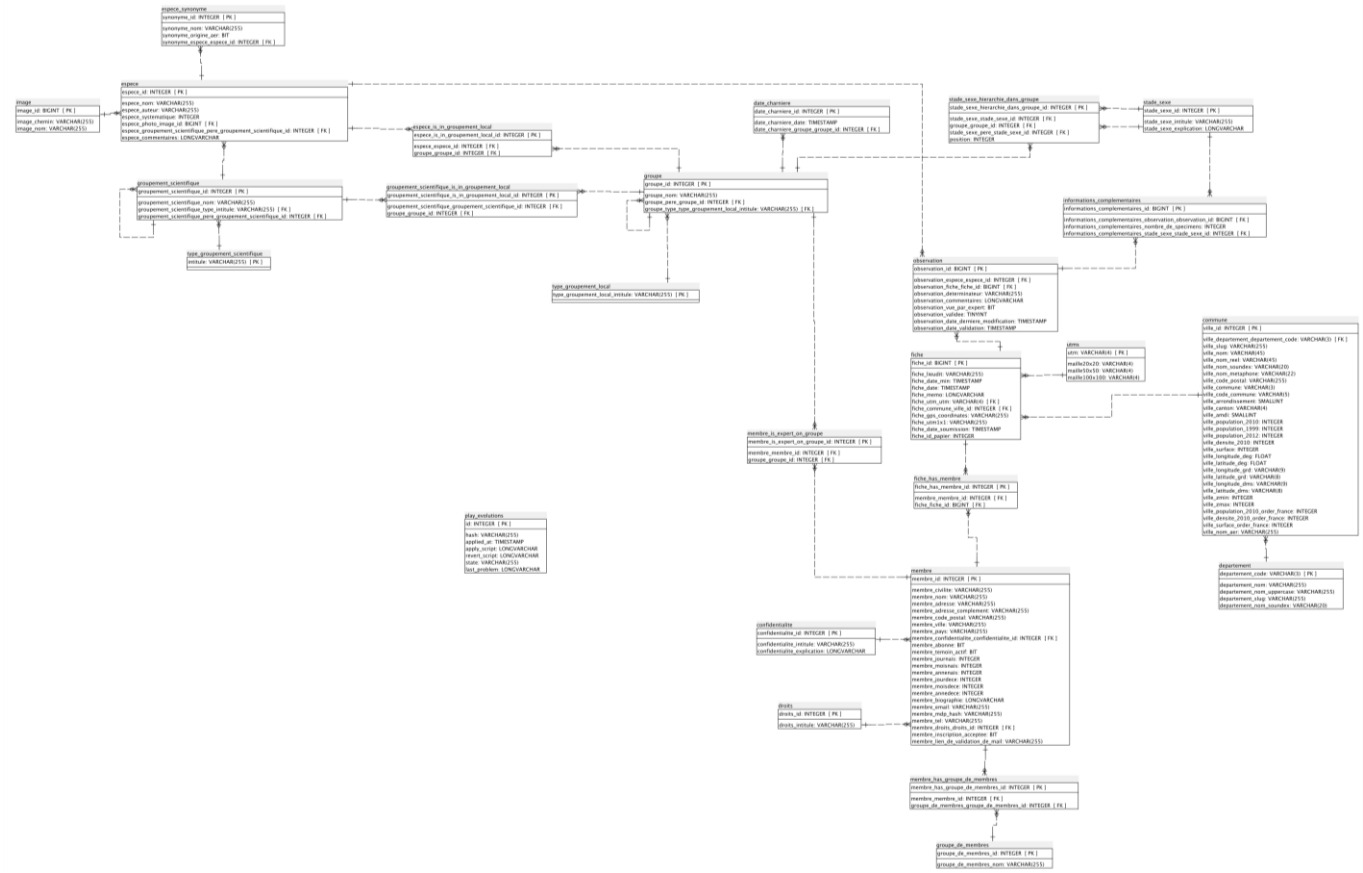
Première séquence d'évènements :

Un membre saisit les données relatives à ses observations de manière incomplète.
Un peu plus tard, ce membre veut modifier ses observations.
Il édite donc son témoignage.
Les experts valident ou invalident les observations dans leur domaine de compétence.
L'utilisateur ne peut plus modifier son observation. S'il le souhaite il doit contacter les experts.

5.4. Schéma de la base de données initiale



5.5. Schéma de la nouvelle base de données



5.6. Algorithme du script de migration

Test de la présence des attributs fiche_utm_utm dans la table **utms** à effectuer avant le script de migration. Non intégré dans le script, fait en amont directement avec SGBD.

Sauvegarde de l'état actuel de la base (pour permettre un retour à un état stable en cas d'erreur)

Début du script de migration

Création des tables **type_groupement_local** et **type_groupement_scientifique** et remplissage avec les données suivantes : groupe, sous-groupe pour la première et ordre, super-famille, famille, sous-famille pour la deuxième

Pour les groupements locaux, on a choisi de garder la table **groupe**. La première idée était de la renommer en **groupement_local**, mais cela entrainait de très nombreuses modifications sur l'application play, et, pressés par le temps, nous avons donc choisi de laisser le nom de la table (ainsi que le nom des attributs), tels qu'ils étaient avant.

Nous avons tout de même ajouté les attributs (et les clés étrangères allant avec) **groupe_type_type_groupement_local_intitule**, et **groupe_pere_groupe_id**, qui n'existaient pas avant, et dont nous avons besoin pour la gestion des groupements locaux.

Mise à jour de la valeur de l'attribut **groupe_type_type_groupement_local_intitule**. Tous les tuples présents dans cette table étant des groupes, cet attribut prend la valeur groupe pour chacun d'eux.

On ajoute ensuite les éléments de la table **sous_groupe** à la table **groupe** (avec le type sous-groupe cette fois-ci). En parallèle, on ajoute leurs anciens et nouveaux identifiants dans une HashMap (les anciens ID sont les clés qui permettent de retrouver les nouveaux facilement, car ils seront nécessaires plus tard dans le script).

On renomme la table **ordre** en **groupement_scientifique**. Ici, on aurait pu créer une nouvelle table, mais nous avons préféré renommer l'ancienne pour réduire

au maximum les déplacements de données. On renomme aussi les attributs existants, bien entendu.

Ensuite, on crée les attributs nécessaires à la gestion des groupements scientifiques qui n'existaient pas dans la table ordre : le type et le groupement scientifique père. On initialise le type à ordre, car il n'y a que les ordres dans la table pour l'instant.

On crée ensuite la table **groupement_scientifique_is_in_groupement_local**. On la remplit à partir des données de **ordre_has_sous_groupe**. On utilise la HashMap créée précédemment pour retrouver le nouvel ID des sous_groupes.

On ajoute les données de la table **super_famille** à la table **groupement_scientifique**. Ici, on fait attention à l'attribut **super_famille_existe** : il ne sert à rien de garder les super familles qui n'existent pas et ne servaient dans l'ancienne base de données qu'à faire le lien entre une famille et un ordre. On crée ici une HashMap pour les super-familles, sur le principe de celle créée pour les sous-groupes : l'ancien ID permet de retrouver le nouveau.

On ajoute ensuite les données de la table **super_famille_has_sous_groupe** à la table **groupement_scientifique_is_in_groupement_local**. On a une nouvelle fois besoin des HashMaps pour retrouver les nouveaux ID des sous-groupes et super-familles.

Pour les familles et les sous-familles, on continue sur le modèle de ce qu'on a fait pour les super-familles. Pour les familles, il faut faire attention à si la super-famille mère existe ou non. Si non, il faut aller chercher l'ordre et le relier à la famille. Pour la sous-famille, tout comme la super-famille, certaines présentes dans la base n'existent pas et ne servent que de lien entre une espèce et une famille. Il faut encore créer des HashMap pour les familles et les sous-familles, toujours sur le même modèle.

Pour les espèces, on commence par créer la table **espece_is_in_groupement_local**, qu'on remplit à partir de deux sources différentes : la table **espece_has_sous_groupe**, et l'attribut de la table **espece** référençant un sous-groupe. Il faut bien évidemment vérifier qu'il n'y a pas de doublons dans la table créée.

On ajoute ensuite un attribut référençant le groupement scientifique père dans une espèce. On met à jour sa valeur pour chaque tuple de la table **espece**, à partir de l'ancien attribut référençant une sous-famille, en faisant attention à l'existence de la sous-famille. Une fois de plus, on utilise les HashMaps sous-groupe, sous-famille et famille.

Ajout de l' attribut **fiche_id_papier** dans la table **fiche**

Création de la clé étrangère sur l'attribut **fiche_utm_utm**

Suppression des anciens attributs et anciennes tables. Voici l'ordre des suppressions :

- les attributs :

-espece_sous_groupe_sous_groupe_id et espece_sous_famille_id dans la table **espece**

-les tables :

-**espece_has_sous_groupe**

-**sous_famille_has_sous_groupe**

-**famille_has_sous_groupe**

-**super_famille_has_sous_groupe**

-**ordre_has_sous_groupe**

-**sous_groupe**

-**sous_famille**

-**famille**

-**super_famille**

Fin du script de migration.

Tests pour voir si la migration s'est bien passée. Sauvegarde du nouvel état de la base de données.

5.6. Planning du projet