Etude Technique Projet Madera





**Projet Madera**

**RILA15**

A.SCHWARZE D.CROCCO C.CHEVALIER G.BOLZINGER

Table des matières

[I. Contexte 3](#_Toc461556830)

[II. Conception générale 4](#_Toc461556831)

[A. Case d’utilisation 4](#_Toc461556832)

[B. Liste des fonctionnalités et hiérarchisation 5](#_Toc461556833)

[C. Règle de gestion 7](#_Toc461556834)

[D. Dépendances internes 8](#_Toc461556835)

[E. Objets métiers 8](#_Toc461556836)

[F. Ergonomie et Mock-up 9](#_Toc461556837)

[III. Conception détaillée 9](#_Toc461556838)

[A. Diagramme de classe 9](#_Toc461556839)

[B. Diagramme de package 10](#_Toc461556840)

[C. Séquence modélisation 11](#_Toc461556841)

[D. Diagramme de composants 12](#_Toc461556842)

[E. Schéma BDD 13](#_Toc461556843)

[IV. Choix des technologies 14](#_Toc461556844)

[A. Type d’application 14](#_Toc461556845)

[Langage de programmation 15](#_Toc461556846)

[Système de gestion de base de données relationnelle 16](#_Toc461556847)

[B. Outil de versionning 18](#_Toc461556848)

[V. Gestion de projet 20](#_Toc461556849)

# Contexte

Introduction

Suite à la validation de la note de cadrage par le Client lors du GO/NOGO qui permettait de redéfinir les besoins du client sous forme d’organigramme, nous allons donc essayer de définir les spécifications techniques de ce projet. Ce document contiendra donc les éléments suivants : Un modélisation UML de l’architecture de la base de données, les dépendances externes, les règles de gestions, la définition de l’ergonomie, des mock-up des différents IHMs, la description des modules et les traitements associés, la définition de l’environnement de développement, les méthodes de développement, ainsi que les choix des technologies qui seront utilisés.

# Conception générale

## Case d’utilisation

Le diagramme des cas d’utilisation va nous permettre de définir les différentes fonctionnalités que va offrir l’application dans le cadre du projet. Ce diagramme gère aussi les différentes contraintes des fonctionnalités.

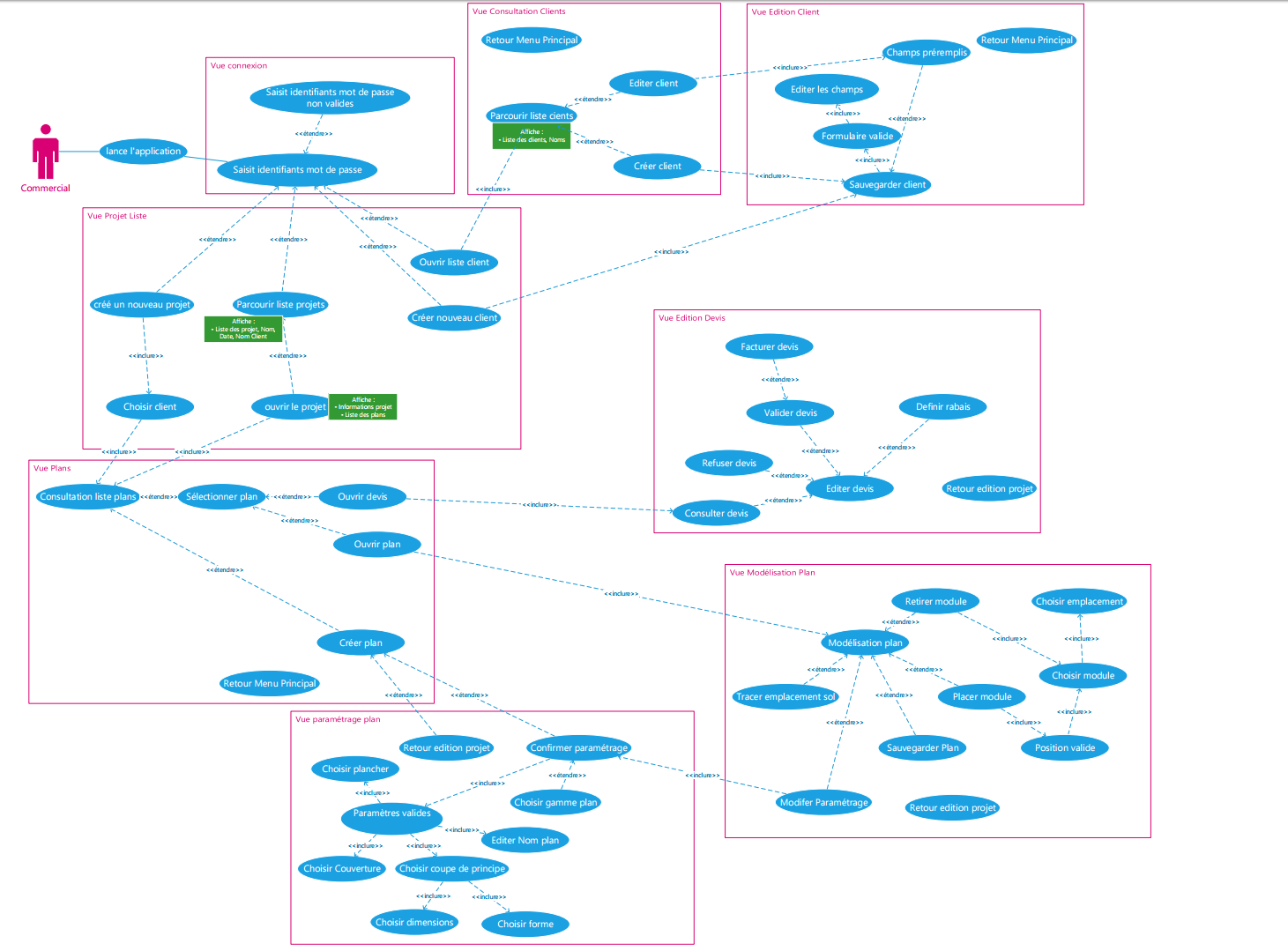


Figure 1 Diagramme de cas d’utilisations

## Liste des fonctionnalités et hiérarchisation

Dans cette partie, nous avons défini les différentes fonctionnalités en deux groupes distincts : les fonctions primaires qui sont essentiels à l’application et les fonctions secondaires qui « constituent » les fonctions primaires.

Liste des fonctionnalités primaires

Ici est établie la liste des fonctionnalités qui « englobe » la totalité des fonctionnalités qui sont détaillées plus bas.

* Identification
* Gestion client
* Gestion projet
* Gestion de devis
* Modélisation de plan

Liste des fonctionnalités secondaires

La liste complète des fonctionnalités répartie selon leur fonctionnalité primaire de rattachement. Le poids représente notre estimation de la charge de développement de chaque fonctionnalité.

|  |  |  |
| --- | --- | --- |
| Fonctionnalité Primaire | Fonctionnalité Secondaire | Poids % |
| Identification |  | 8 |
|  | Tester la connectivité | 2 |
|  | Se connecter | 3 |
|  | Afficher erreur champs | 1 |
|  | Se déconnecter | 2 |
| Gestion client |  | 12 |
|  | Afficher la liste des clients | 4 |
|  | Afficher les infos client | 2 |
|  | Créer un client | 2 |
|  | Editer un client | 3 |
|  | Afficher erreur champs | 1 |
| Gestion de projet |  | 20 |
|  | Afficher la liste des projets | 1 |
|  | Créer un projet | 1 |
|  | Ouvrir un projet | 2 |
|  | Récupérer la liste des clients | 1 |
|  | Afficher les infos d’un projet | 1 |
|  | Afficher la liste des plans d’un projet | 3 |
|  | Créer un plan | 2 |
|  | Ouvrir un plan | 2 |
|  | Afficher les infos d’un plan | 1 |
|  | Consulter le devis | 3 |
|  | Copier le plan | 1 |
|  | Charger données paramètres | 1 |
|  | Afficher erreur champs | 1 |
| Gestion de devis |  | 25 |
|  | Charger composants | 4 |
|  | Afficher infos plan | 1 |
|  | Charger données client | 2 |
|  | Charger données commercial | 2 |
|  | Calculer les prix | 5 |
|  | Exporter | 1 |
|  | Appliquer une remise | 2 |
|  | Afficher un dossier technique | 3 |
|  | Modifier l’état du devis | 4 |
|  | Afficher erreur champs | 1 |
| Modélisation |  | 35 |
|  | Editer les paramètres éditables | 1 |
|  | Sauvegarder le plan | 2 |
|  | Vérifier les contraintes du plan | 1 |
|  | Quitter | 1 |
|  | Tracer un slot | 2 |
|  | Retirer un slot tracé | 1 |
|  | Sélectionner un slot | 3 |
|  | Choisir un module | 5 |
|  | Sélectionner un module | 3 |
|  | Sélectionner un slot d’un module | 4 |
|  | Choisir un module pour un slot de module | 5 |
|  | Mettre à jour l’affichage | 5 |
|  | Charger les modules compatibles | 1 |
|  | Charger les composants d’affichage | 1 |

## Règle de gestion

Les règles de gestion que nous avons définies vont permettre à la fois de limiter les erreurs que l’utilisateur peut provoquer ainsi que d’augmenter la sécurité des données dans l’application.

Elles sont les suivantes :

Règle Client :

* Un nom de client ne peut contenir des caractères spéciaux ou des chiffres
* Un client possède un commercial référent, mais ses projets peuvent très bien être gérés par un autre commercial en cas d’absence ou autres
* Un client peut créer autant de plans qu’il souhaite
* Un client ne peut posséder qu’un seule et unique ficher client
* L’ajout d’un client avec les mêmes attributs qu’un autre client ne peut être effectué

Règle Modélisation :

* La création d’un slot à l’extérieur des murs de la maison n’est pas possible
* La suppression des slots des murs du plan de base n’est pas possible
* Un plan ne peut être sauvegardé uniquement si les recouvrements des murs extérieurs sont choisis
* Le choix des paramètres du plan ne peut être enregistré si les champs obligatoires (Nom, choix de la base, choix du plancher, choix du ne sont pas renseignés
* Le choix de la gamme n’est pas un choix obligatoire

Règle Devis :

* L’état d’un devis ne peut pas revenir sur un état précédent
* Le prix total d’un devis est fixé le jour de la génération du devis et ne peux évoluer.
* Un devis refusé ne peut être validé
* L’offre de remise d’une gamme n’est appliquée si et seulement si tous les modules du plan font partie de cette gamme
* L’application de remise, marge, rabais n’est possible que sur un devis à l’état brouillon

Règle Projet :

* Un projet ne peut appartenir qu’à un seul et unique client
* Un commercial peut reprendre un projet déjà existant
* Les dates seront sélectionnables via un calendrier et ne peut être écrit à la main.

## Dépendances internes

Les dépendances internes sont les différents contraintes qui pourraient ralentir ou même retarder le projet, c’est pour cela que nous avons pris soin de les définir. Voici la liste :

* Communication avec serveur de base de données du Bureau d’études :
  + Compte utilisateur avec accès en lecture
  + Adresse et port du serveur de base de données
* Base de données du Bureau d’études remplies et fonctionnelle
* Formation technique des utilisateurs (15 commerciaux) :
  + à l‘utilisation du support matériel
  + à l’utilisation de l’application Madera MMB
* Equipe de développement opérationnelle pendant toute la durée du développement
  + 4 développeurs de septembre 2016 à avril 2017
* Chef de projet informatique disponible à en cas d’incident lors des phases 3 et 4
* Matériel prévu disponible pour la phase de recettage

## Objets métiers

Dans cette partie, nous avons défini les différents objets participants au projet. Nous l’avons modélisé sous forme de diagramme d’objet comme ci-dessous

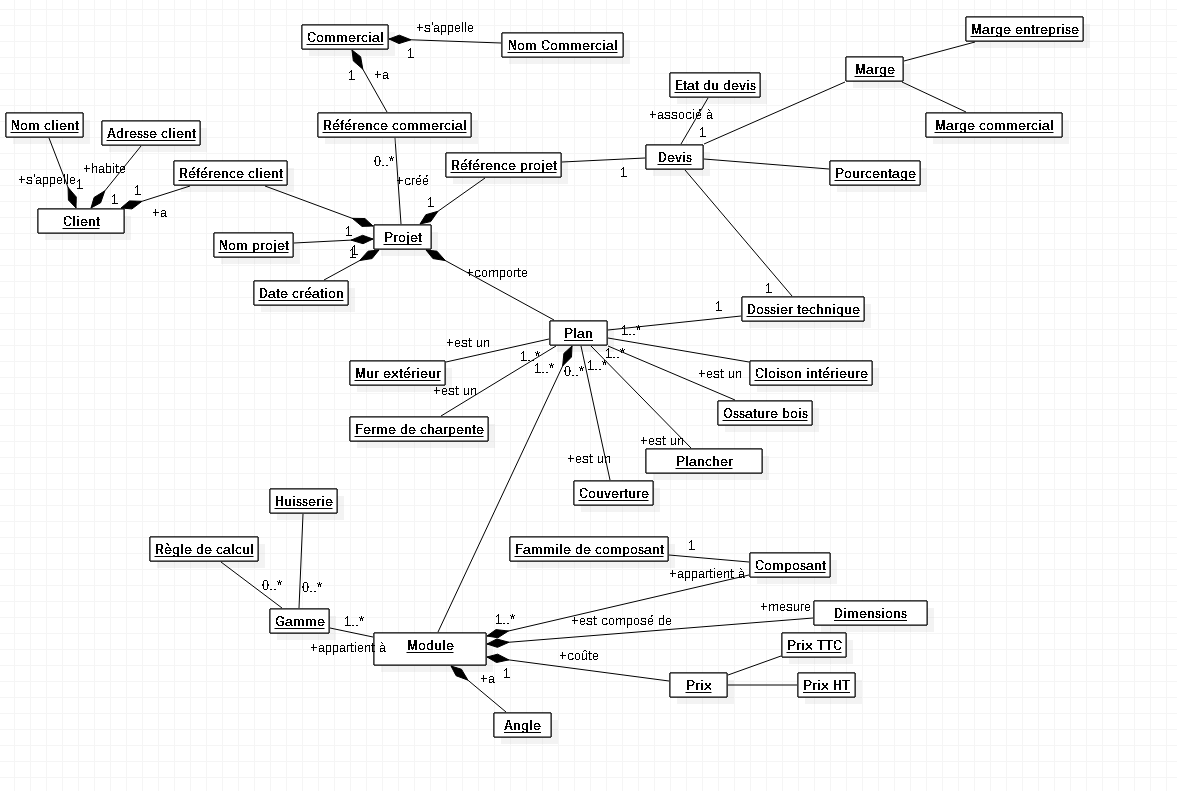


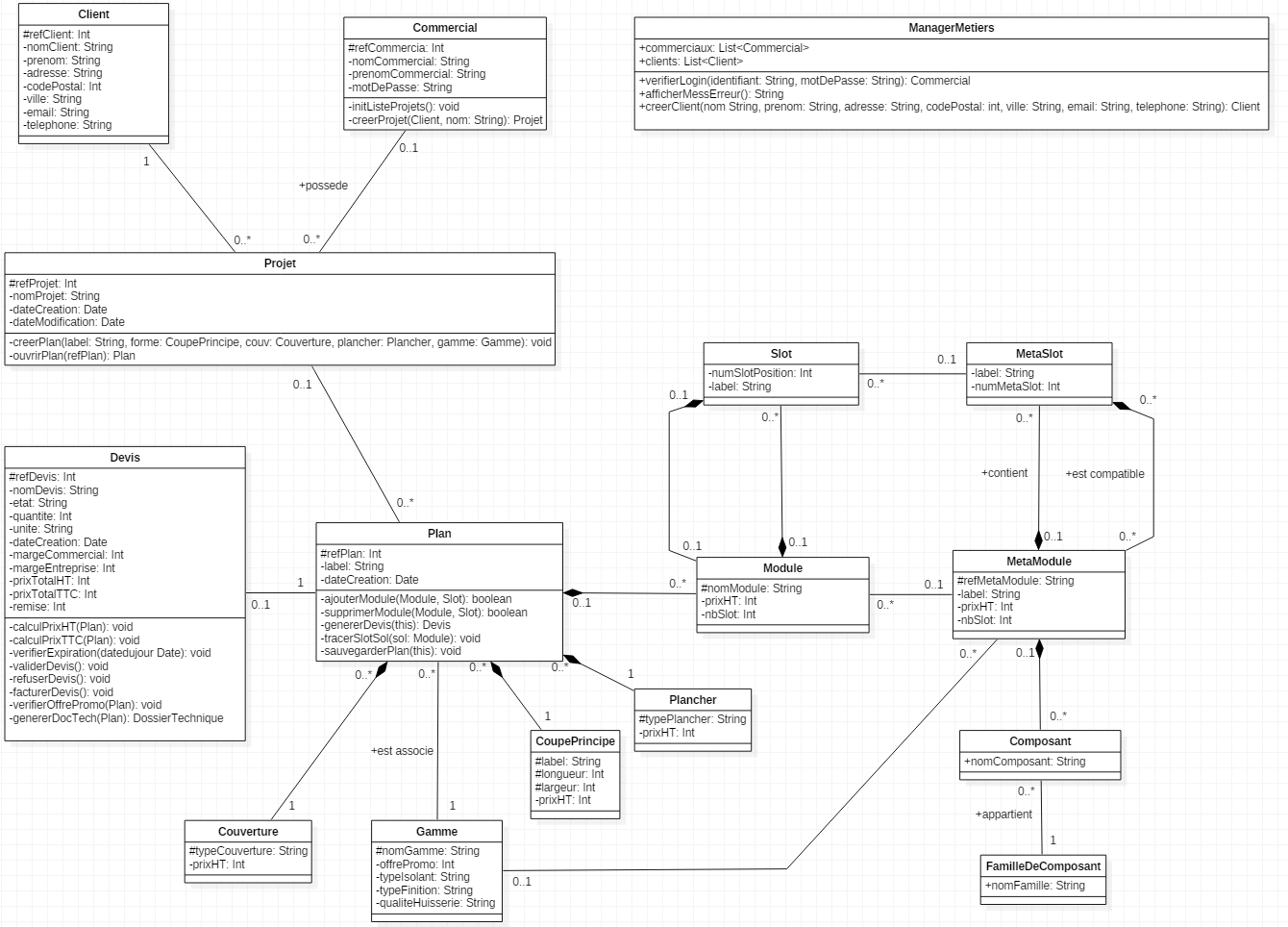
Figure 2 Diagramme d'objet

## Ergonomie et Mock-up

Pour avoir un visuel des différents IHM que nous allons développer dans le livrable 3, nous avons défini des critères d’ergonomie, tels que un interface souple, simple d’utilisation et performant. Nous avons par ailleurs créer les différents mock-up de chacuns des différents IHM qui sera présent dans l’application. (Ils sont disponibles en Annexe)

# Conception détaillée

## Diagramme de classe

Cette partie concernera toute la conception détaillée de l’application notamment avec des diagrammes UML. Ils permettront de modéliser la partie applicative et la partie base de données.

Dans un premier temps, nous avons réalisés un diagramme de classe répertoriant toutes les classes comme le montre l’image ci-dessous.

Figure Diagramme de classe

## Diagramme de package

Pour structurer le diagramme de classe et avoir une meilleur visibilité ainsi qu’une meilleur organisation lors de la phase de développement, nous avons modélisé un diagramme de package qui regroupe les différentes classes.

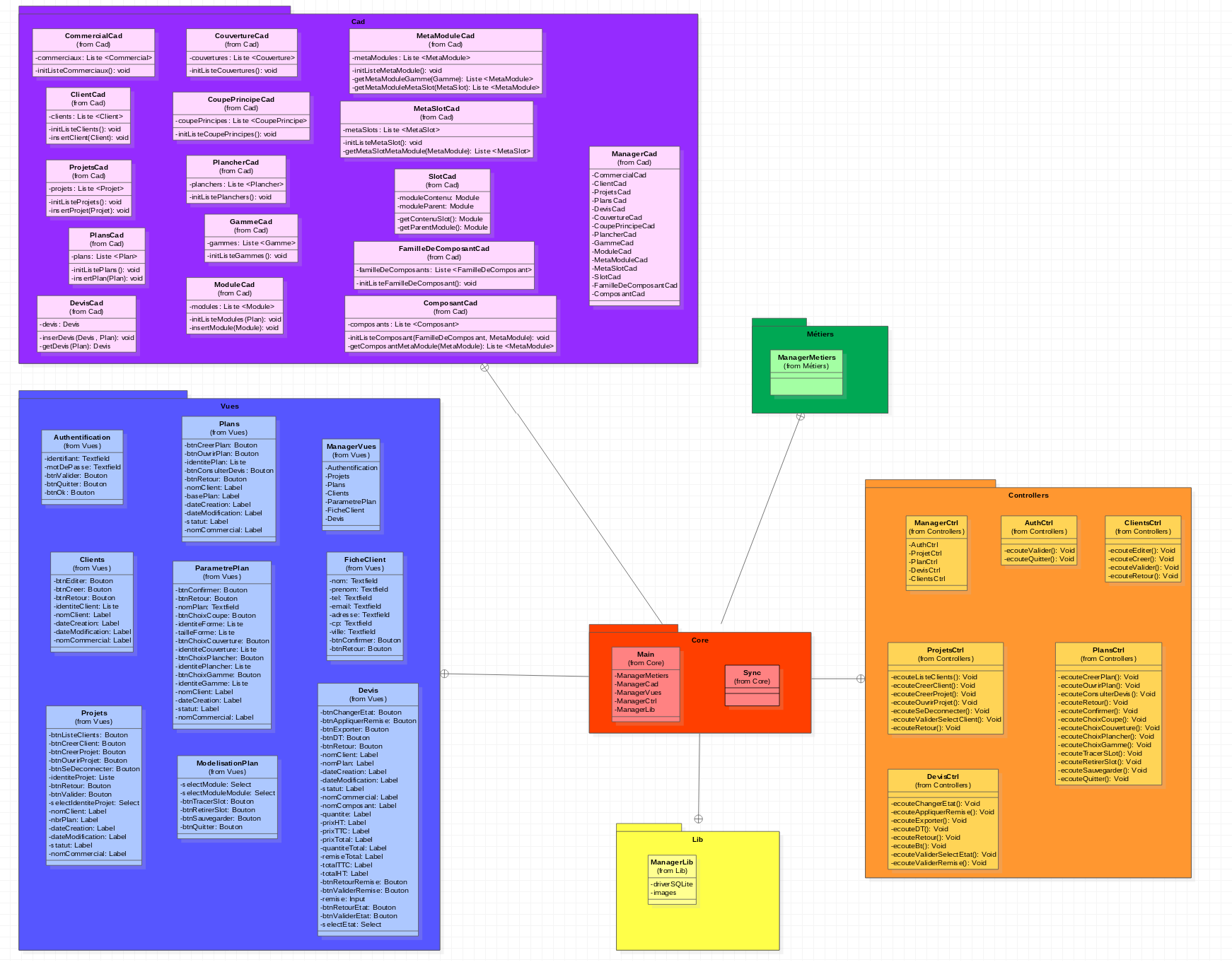


Figure 4 Diagramme de package

## Séquence modélisation

Après avoir défini le diagramme de package, nous pouvons dès à présent définir les différentes interactions entre acteurs et le système. Pour cela nous avons réalisé plusieurs diagrammes de séquence pour chaque fonctionnalité primaire. Voici un exemple d’un diagramme de séquence pour la fonctionnalité permettant d’ajouter un module à un slot. (Les autres diagrammes seront disponibles en annexes)

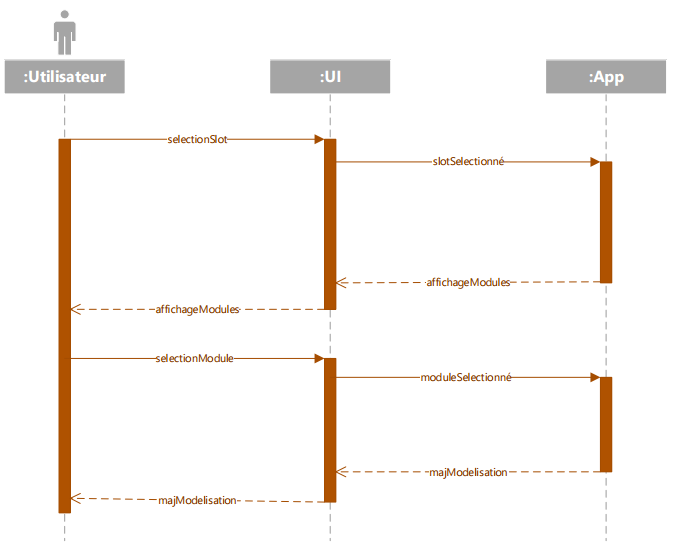


Figure 5 Diagramme de séquence

## Diagramme de composants

Suite à la modélisation du diagramme de package ci-dessus, le but était de modéliser les différents composants qui aller interagir dans l’application dans le cadre du projet. Pour cela nous avons réalisé un diagramme de composants.

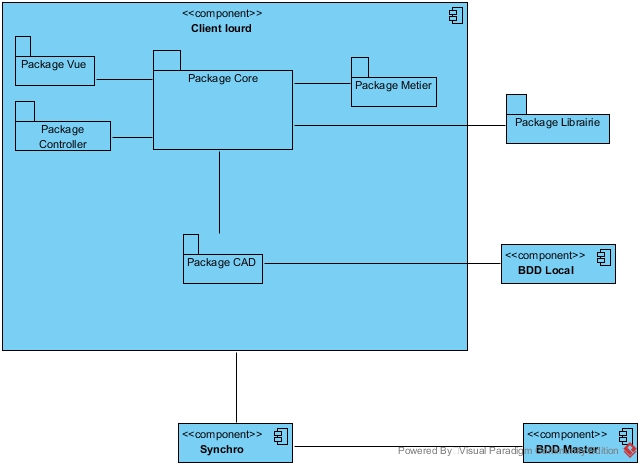


Figure 6 Diagramme de composants

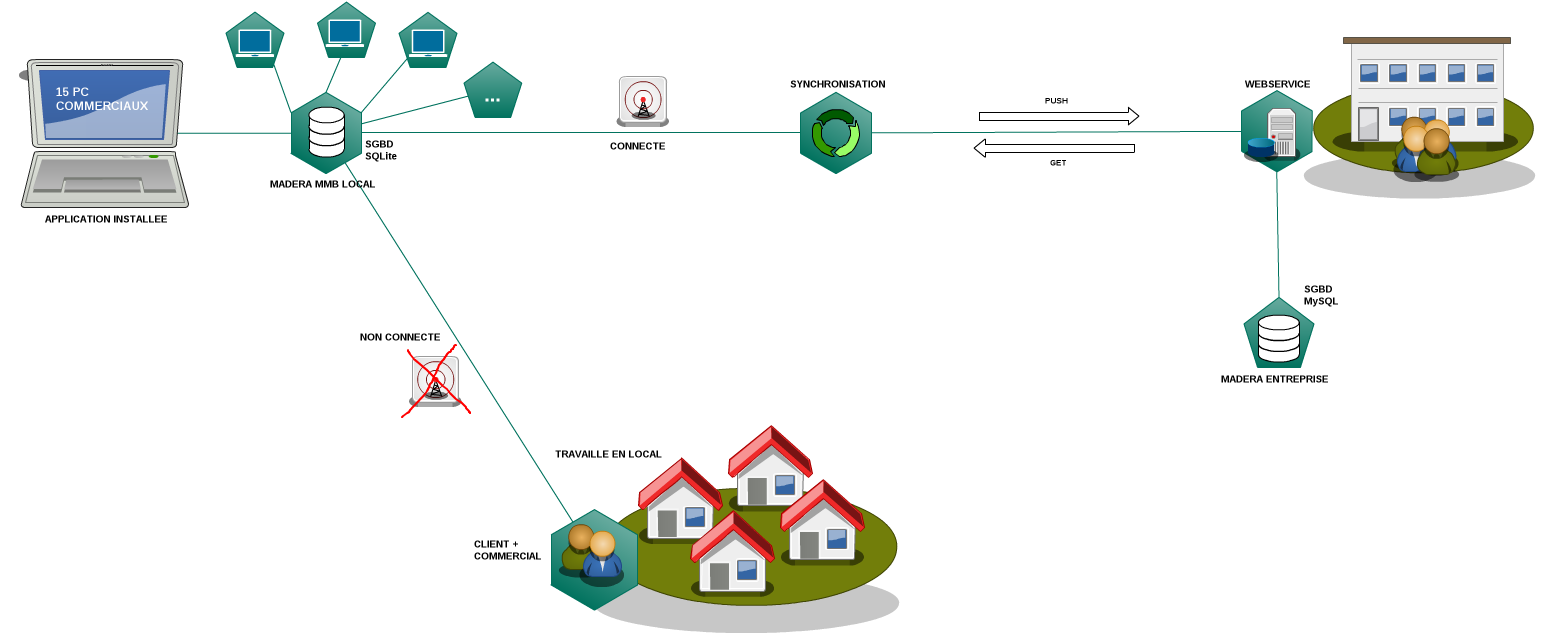
Comme vous pouvez le voir Nous avons un client lourd qui communique avec une base de données local via le package CAD (qui fera la transmission de données entre les deux). Le client lourd lors de l’utilisation des IHM utilisera le package Librairie. La base de données en local sera alimentée par la BDD Master via une synchronisation et un web service REST pour des questions de sécurité. Lors que l’utilisateur est en mode « hors-ligne » il utilisera sa base de données en local. Et du moment, qu’il passera en mode « connecté », la synchronisation s’effectuera en mettant à jour (si nécessaire) la base de données. Pour donner un exemple plus visuel voici un schéma de notre architecture.

Figure 7 Schéma de l'architecture du projet

## Schéma BDD

La modélisation UML nous aura permis de définir un diagramme de classe qui va permettre ensuite de définir le schéma de la base de données qu’utilisera l’application. Voici le schéma base de données ci-dessous.

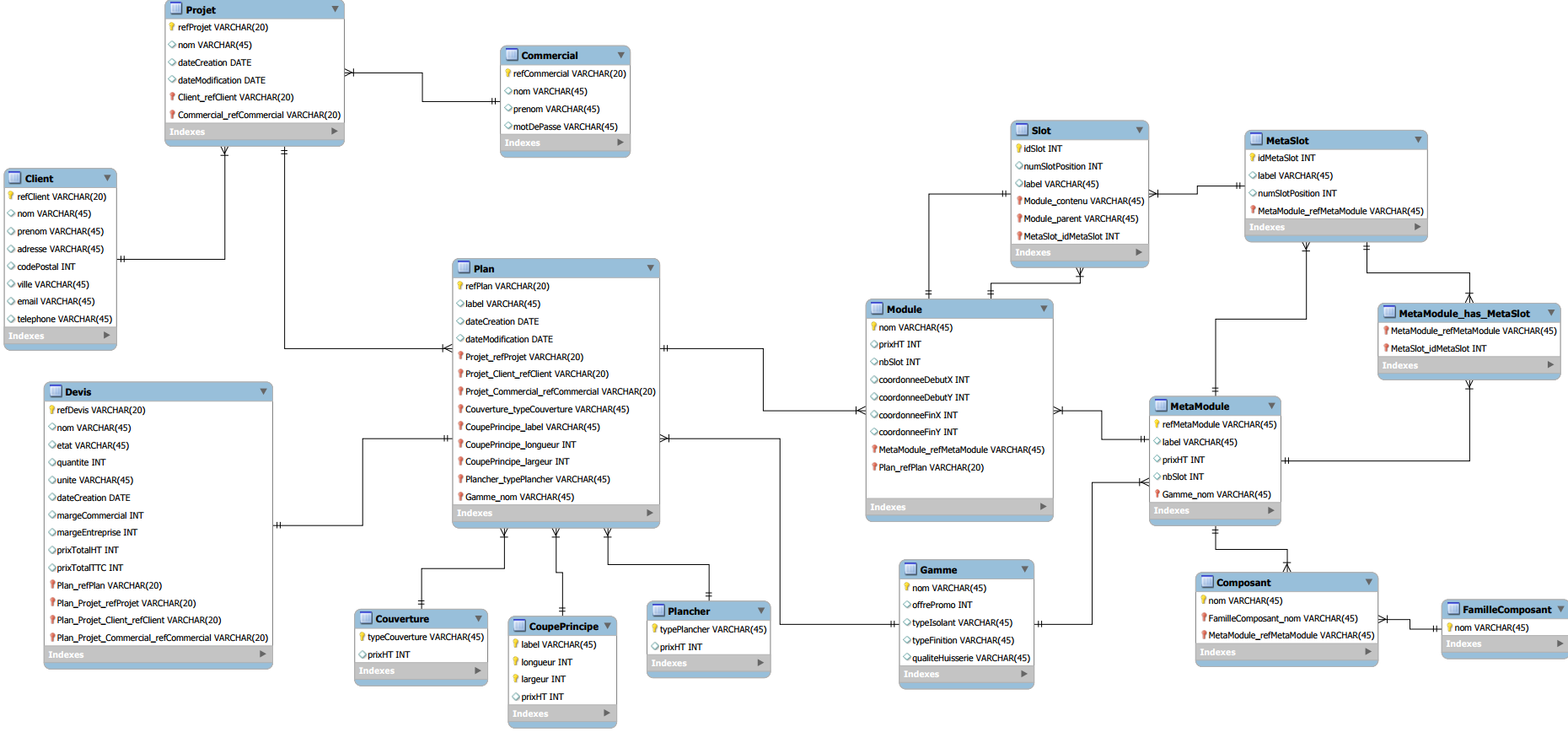


Figure 7 Schéma BDD

# Choix des technologies

## Type d’application

Client Lourd

**Avantages :**

* Un seul langage client et serveur
* Communication avec le serveur simplifié
* Rapidité de production
* Plateformes d’exécution rétro-compatibles
* Peut évoluer en client riche pour minimiser les problèmes de déploiement et de maintenance

**Inconvénients :**

* Déploiement plus lourd
* Maintenance et Mises à jour plus lourdes
* Sécurité et sauvegarde dépendantes du poste client

Client Léger

**Avantages :**

* Coût plus faible dans le cadre d’un parc info
* Maintenance système, application centralisée et immédiatement disponible après chaque mise à jour
* Maintenance utilisateur
* Robustesse, serveur centralisé moins sensible aux pannes qu’un poste client
* Mobilité

**Inconvénients :**

* Langages client/serveur différents
* Partie cliente nécessitant plusieurs langages
* Formation des développeurs à l’intégralité de la pile technique de l’application
* Pas de mode hors-ligne, dépendant de la qualité du réseau
* Evolutions des navigateurs
* Performances en baisse si trop nombreux utilisateurs

Conclusion : Client lourd

Nous avons donc décidé d’opter pour un client lourd pour les raisons suivantes :

* Le parc informatique dédié à la vente des maisons modulaires consistera en seulement en quinzaine d’ordinateurs portables ce qui réduira les inconvénients de déploiement et de maintenance
* L’application Madera MMB ne nécessite aucun travail collaboratif
* L’application Madera MMB nécessite d’être utilisée en mode hors connexion
* La rapidité de production d’un client lourd offrira une sécurité dans le respect du planning de développement
* Les AGL pour clients offre offrent des modules permettant le travail collaboratif

## **Langage de programmation**

Pour le choix du langage, nous avons mis en place une grille décisionnelle pour nous aider à déterminer le langage le plus adéquat selon une liste de critères

Pour développer un client lourd notre choix s’est porté entre deux langages : C# et Java

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Critères | Importance | Note C# | Note Java |  |  |  |
| Connaissance du langage | 5 | 2 | 5 |  |  |  |
| Sécurité des données | 2 | 3 | 4 |  |  |  |
| Pérennité | 2 | 5 | 4 |  | **Légende** | **Note** |
| Performance | 3 | 4 | 3 |  | Elevé | 5 |
| Portabilité | 3 | 4 | 2 |  | Haut | 4 |
| Faisabilité | 5 | 4 | 2 |  | Moyen | 3 |
| Déploiement | 3 | 3 | 3 |  | Bas | 2 |
| Cout | 4 | 1 | 3 |  | Faible | 1 |
| Maintenabilité | 3 | 4 | 2 |  |  |  |
| Stabilité | 3 | 4 | 4 |  |  |  |
| Popularité | 5 | 5 | 4 |  |  |  |
| Normalisation | 3 | 5 | 4 |  |  |  |
| Utilisation | 3 | 3 | 4 |  |  |  |

**Total**  **47 44**

Il en sort que le C# sera le langage choisi du fait de son obtention de la note la plus élevée.

Afin de séparer la conception graphique de l’interface et des traitements de l’application, et pour sa compatibilité native avec le langage C#, nous avons donc choisi d’utiliser XAML pour la partie graphique.

Ainsi l’application sera développée en C#/XAML.

## **Système de gestion de base de données relationnelle**

**Choix SGBD ?**

Dans notre configuration proposée, nous devons effectuer un choix sur les bases de données. A savoir, que d'après le schéma, nous envisageons de mettre en place, une base de données dites

« Embedded », c'est à dire qui sera intégré localement et utilisable sur une machine par l'utilisateur, ainsi qu'une seconde base dites « master », qui sera la base de données principal de l'entreprise

MADERA. Nous prévoyons d'effectuer des synchronisations automatisées à chaque démarrage de l'application. En mode connecté, les données stockées sur la base de données master seront synchronisé sur la base de données locale. En mode sans connexion, l'application pourra fonctionner simplement avec la base de données locale et y stocker les nouvelles données qui seront créés.

**➢ Un coup d'œil sur les différents types de SGBD No SQL ?**

On peut classer celle-ci en 4 catégories ou 4 grandes familles :

1. les bases de données orientées doc : MongoDB, CouchDB, RavenDB, MarkLogic …

2. les bases clé/valeur : Aerospiken, Redis, Riak, Voldemort ...

3. les bases en colonnes : Accumulo, Cassandra, Hbase, HyperTable ...

4. les bases orientées graphes : InfiniteGraph, Neo4j, FlockDB, Giraph …

Les bases de données orientées document sont idéalement adapté pour les applications en évolution permanente. L'ensemble des données sont décrites dans une structure de document au format JSON ou XML dans la plupart des cas. On peut interroger la BDD sur tous les composants du schéma défini. L’avantage est de pouvoir récupérer, via une seule clé, un ensemble d’informations structurées de manière hiérarchique. Pas de jointures.

Les bases clé/valeur offrent un bon niveau de performance lorsqu'il s'agit de faire des recherches sur des jeux de données relativement simple. On ne peut interroger la BDD que sur sa clé. Bien adapté aux applications nécessitant un accès rapide aux données.

Les bases en colonnes offrent un haut niveau de performance lorsqu'il s'agit de traiter des données à grande échelle. Le nombre de colonnes est dynamique. Idéalement adapté au stockage de listes

(Messages, billets, commentaires ...)

Les bases orientées graphes offrent une solution qui privilégie les relations entre les valeurs et stocke les données en utilisant une structure des graphes en nœuds, en relations et en propriétés. Ce type de base de données est surtout utilisé pour les mises en œuvre à l'échelle du Web, du Big Data et à caractère analytique. Pour simplifier, elles permettent de gérer des relations multiples entre des objets.

**➢ SQLite – les avantages ?**

Licence : OpenSource et gratuit

Serveur : Pas de serveur

SQL : moteur SQL simplifié

Disponibilité : Linux, MacOSX, Windows, Unix, BSD

Environnement : Multiplate-forme

Installation & Configuration : Très facile à installer - 0 configuration requise ou presque

Type : Système de gestion de base de données relationnelle

Migration : Possibilité de passé à MySQL par la suite

Synchronisation : Plus facile lorsqu'il y a 2 BDD SQL

Fonctionnalités : Majeure partie des fonctionnalités classiques.

Version : La dernière version SQLite3 offre plus de performance (auto incrément, meilleur stockage, support clés étrangères)

Mais aussi : Il est portable, performant surtout en lecture et très simple. C'est à dire, qu'il y a aucune manipulation à faire. La base est stockée dans un seul fichier. Elle est incluse directement dans l'application.

**➢ SQLite – les inconvénients ?**

Fonctionnalités : Ils manquent certaines fonctionnalités importantes telles qu’ALTER TABLE,

GRANT, REVOKE, …

Limitations : Pas de reconnaissance de transactions imbriquées ni les « triggers ». Pas de réplication.

Sécurité : Minimum. Pas de notion de rôles, de hiérarchisation de groupes.

**Conclusion :**

Nous avons donc en définitive choisi **SQLite** pour la gestion de la base de données locale à l’application et sera donc la base de donnée « esclave » de la partie client.

La partie serveur, aura la charge d’héberger la base de donnée du bureau d’études.

Celle-ci, contenant un plus gros volume de données, devra être gérée sous **MySQL** . Cela permettra d’utiliser les fonctionnalités de rôles et de hiérarchisation des groupes pour administrer

Atelier de génie logiciel

* Pourquoi un AGL ?

Pour optimiser la productivité dans le développement de l’application en C# XAML, il nous a paru nécessaire de travailler avec un outil permettant de documenter automatiquement le programme, et de maintenir en permanence à jour cette documentation, et, ce tout au long de sa conception.

L’utilisation d’un AGL facilitera la collaboration entre les différents programmeurs ainsi que la maintenance ultérieure de la solution en partageant les mêmes méthodes.

* Pourquoi Visual Studio ?

Plateforme de développement logiciel édite par Microsoft, Visual Studio est un ensemble complet d'outils de développement permettant de générer, entre autres, des applications bureautiques et de partager plusieurs technologies telles que C# et XAML, nous concernant.

Il intègre de nombreuses fonctionnalités telle que le travail collaboratif et permet une production rapide des applications bureautique.

Visual Studio se trouve donc être le candidat parfait pour devenir notre AGL dans le développement de l’application Madera.

* Pour quoi 2013 ?

Visual Studio 2013 est connu à ce jour pour être l’une des versions les plus stables de Visual Studio. Elle offre la possibilité de développer des applications multiplateformes. Fonctionnalité que ne propose pas la version précédente.

Egalement, Visual Studio 2015 est une version réputée pour ne pas être stable et offrant des fonctionnalités dont notre équipe n’a pas l’utilité.

**Conclusion :**

Notre équipe utilisera donc l’AGL Visual Studio 2013 dans le développement de l’application MaderaMMB.

## Outil de versionning

On adoptera une méthode particulière de travail (workflow) avec git. Cela a pour but de structurer le développement.

Il faut se rappeler que Git fonctionne avec un système de branche :

(master) # git checkout -b newbranch

(newbranch) #

modifications...

(newbranch) # git add ...

(newbranch) # git commit -m "modifications"

(newbranch) # git checkout master

(master) #

(master) # git merge newbranch

(master) # git branch -d newbranch

Branche d'intégration : master/dev

Dans la branche **master**, rien ne doit être committé. Cette branche sert pour les releases. La branche **dev** elle fait office de branche d'intégration (intégration des fonctionnalités, des bugfix, des hotfix, ...).

Branche de fonctionnalités

Le développement d'une fonctionnalité se fait dans un branche issue de la branche **dev** :

(dev) # git checkout -b feature-mafeature

(feature-mafeature) #

modifications...

commits...

Une fois la fonctionnalité testée, et fonctionnelle, on l'intègre à la branche **dev** :

(feature-mafeature) # git checkout dev

(dev) # git merge --no-ff feature-mafeature

(dev) # git branch -d feature-mafeature

Si la branche a été pushé sur le remote :

(dev) # git push origin :feature-mafeature

git merge --no-ff feature-mafeature, ici le --no-ff permet de forcer la création d'un nouveau commit lors du merge. Lorsqu'un merge à lieu, et qu'il n'y a aucun conflit, un **fast-forward** à lieu, et fusionne l'historique des deux branches. Mais lorsqu'il y a un conflit, un nouveau commit est créé contenant les modifications pour régler le conflit.

Ici, on veut forcer la création de ce commit (il sera donc vide s'il n'y a pas de conflit). Ainsi dans le graphique de l'historique on voit bien le début de la branche de feature, jusque son merge. On peut ainsi déterminer à partir de quelle intégration un bug a pu survenir.

Branche de bugfix

De la même manière, lorsqu'un bug est présent, on créé une nouvelle branche afin de le fixer :

(dev) # git checkout -b bugfix-monbugfix

(bugfix-monbugfix) #

modifications...

commits...

(bugfix-monbugfix) # git checkout dev

(dev) # git merge --no-ff bugfix-monbugfix

(dev) # git branch -d bugfix-monbugfix

Si une feature était en cours de développement :

(dev) # git checkout feature-mafeature

(feature-mafeature) # git merge dev

Branche de release

Lors de la préparation d'une release, on créé une nouvelle branche, issue de la branche **dev** :

(dev) # git checkout -b release-1.2

(release-1.2) #

modification du numéro de version

commit

Pour des bugfixs de dernière minute on les intègre comme d'habitude dans **dev** et on les merge dans la branche de release :

(dev) # git merge --no-ff bugfix-monbugfix

(dev) # git checkout release-1.2

(release-1.2) # git merge --no-ff bugffix-monbugfix

(release-1.2) # git checkout dev

(dev) # git branch -d bugfix-monbugfix

Une fois la release prête à la distribution, on l'intègre dans la branche **master** :

(release-1.2) # git checkout master

(master) # git merge --no-ff release-1.2

(master) # git tag -a 1.2

(master) # git checkout dev

(dev) # git merge --no-ff release-1.2

(dev) # git branch -d release-1.2

Branche de hotfix

Il arrive qu'après la sortie d'une release, un bug intervient, un bug critique, devant être résolu immédiatement, on va donc créer une branche de hotfix issue de la branche **master**:

(master) # git checkout -b hotfix-1.2.1

(hotfix-1.2.1) #

modifications...

commits...

(hotfix-1.2.1) # git checkout master

(master) # git merge --no-ff hotfix-1.2.1

(master) # git tag -a 1.2.1

(master) # git checkout dev

(dev) # git merge --no-ff hotfix-1.2.1

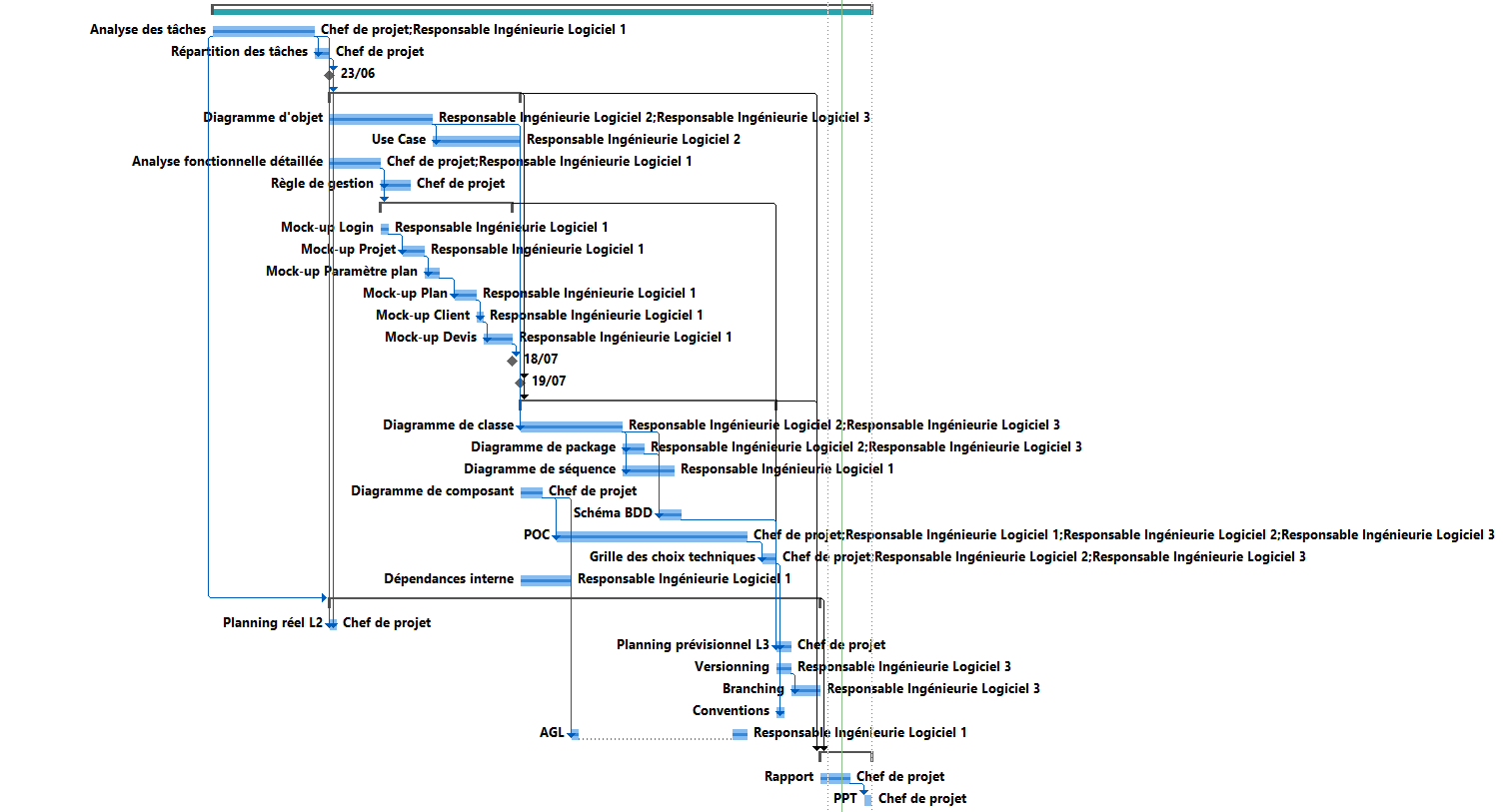
(dev) # git branch -d hotfix-1.2.1

Conclusion

On a donc un développement structuré :

* la branche **master** est taggué à chaque nouvelle version (cela génèrera un archive **zip** et **tgz**, téléchargeable sur **github**), et ne contient que des merges ;
* la branche **master** ne contient que des versions stables ;
* la branche **dev** ne contient également que des merges des autres branches ;
* la branche **dev** contient un contenu stable, dans le cas contraire les bugfixs sont directement intégrés dans cette branche, elle possède également toutes les dernières features ;
* la branche **dev** n'est jamais corrompue par les features/bugfixs en cours de développement.

# Gestion de projet



Voici notre planning de gestion de projet concernant ce livrable 2 : Etude technique. (Le MsProject est disponible en annexe pour plus de détails). Les différents comptes rendus de réunion seront consultable dans les annexes

Figure Planning Livrable 2

Pour définir la méthodologie de développement du livrable 3, nous avons créé une matrice des itérations pour définir quels modules est prioritaires et les différents packages et IHM qui seront développés en amont.

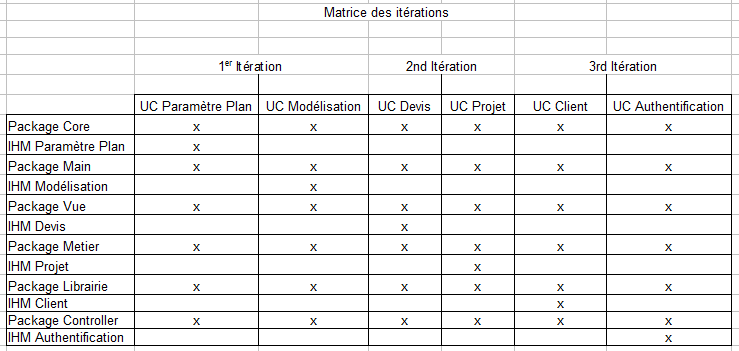


Figure 9 Matrice des itérations

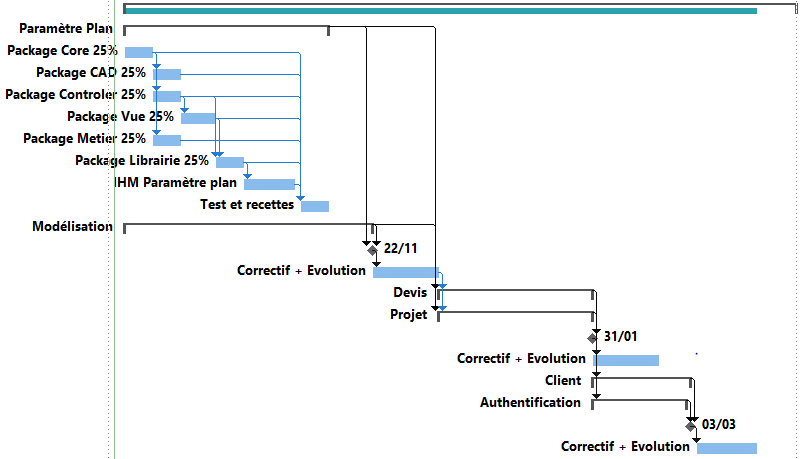
Et voici le planning prévisionnel pour le livrable 3  qui concerne la phase de développement du projet (Qui sera lui aussi disponible en Annexe)

Figure Planning prévisionnel

**Conclusion du chef de projet :**

Ce livrable s’est bien déroulé malgré des retards et des difficultés. Nous avons su être soudés et complémentaires pour réaliser tous les livrables demandés. Néanmoins la gestion de l’humain est très difficile à gérer. C’est la chose qui a été la plus dure durant ce livrable pour ma part (Réussir à organiser des réunions avec tous les membres, en dehors des horaires de travails relève parfois du défis). Je suis tout de même satisfait de la quantité et de la qualité du travail que l’on a fourni pour ce livrable. Nous avons privilégié les prises de décisions en groupe avec notamment des plannings pockets et des débats ouverts et des méthodes AGILE.

Nos principales difficultés ont été de nous mettre d’accord. Nous avons perdus beaucoup de temps à nous entendre sur certains points.