***Dossier architecture logiciel***

Le directeur du restaurant « Au délice de Bigoune » a décidé d’investir dans le développement d’une application de gestion et supervision du fonctionnement de son restaurant.

Application de gestion et supervision

**Groupe 1**

**05/12/2018**

Antoine Savalle

Nicolas Van-Hove

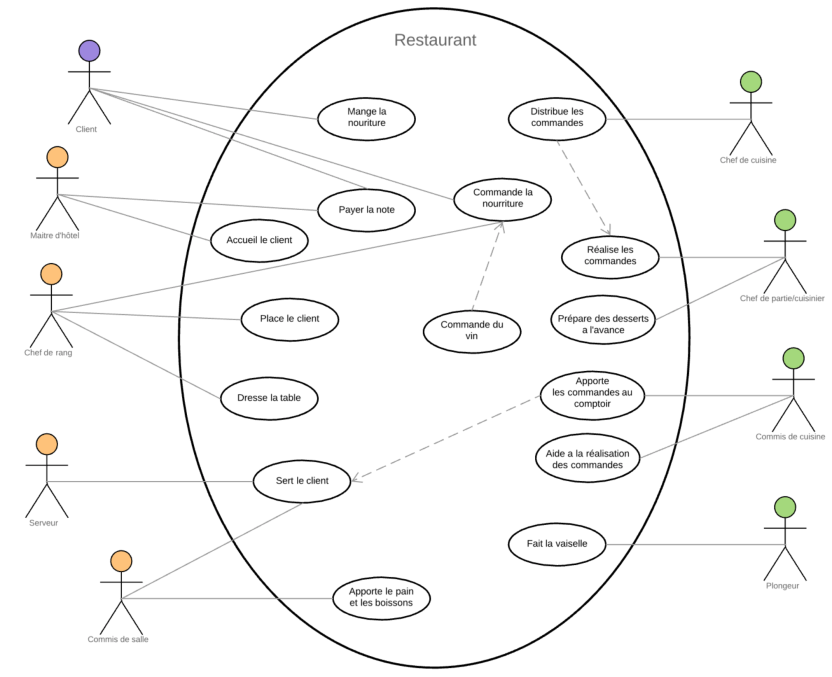
Maxime Mulder

Elian Hamont

Pierre Hamel

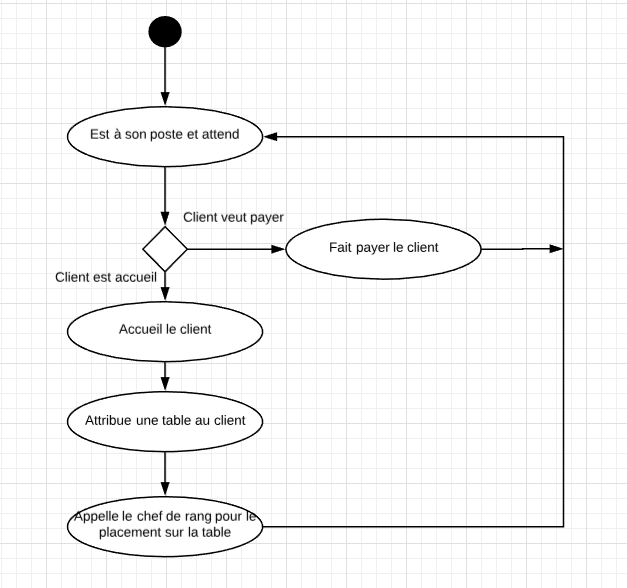
***Sommaire***

1. Diagrammes UML (technique et fonctionnels)
   1. Diagrammes de cas d’utilisation
   2. Diagramme d’activité
   3. Diagrammes de composants et classes
   4. Diagrammes de séquences
2. Justifications et détails des DP utilisés
   1. MVC :
   2. Singleton :
   3. Factory :
   4. Observer :
   5. DAO :
   6. DTO :
   7. Injection de dépendances :
3. Base de données
   1. MCD
   2. MLD
   3. Script SQL
4. ***Diagrammes UML (techniques et fonctionnels) =***
   1. Diagrammes de cas d’utilisation :

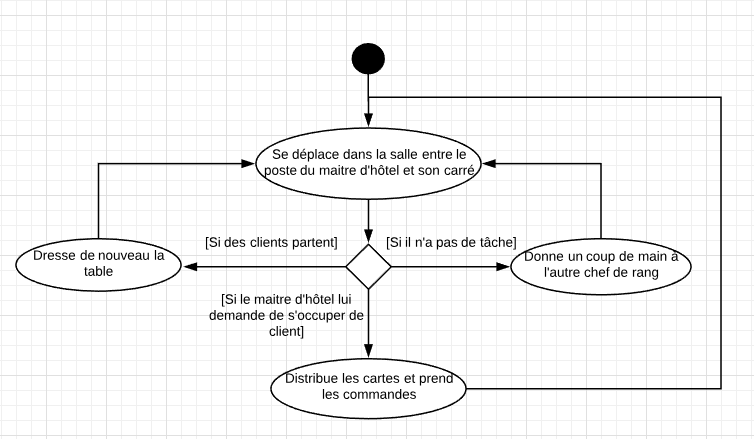
****

* 1. Diagrammes d’activité :

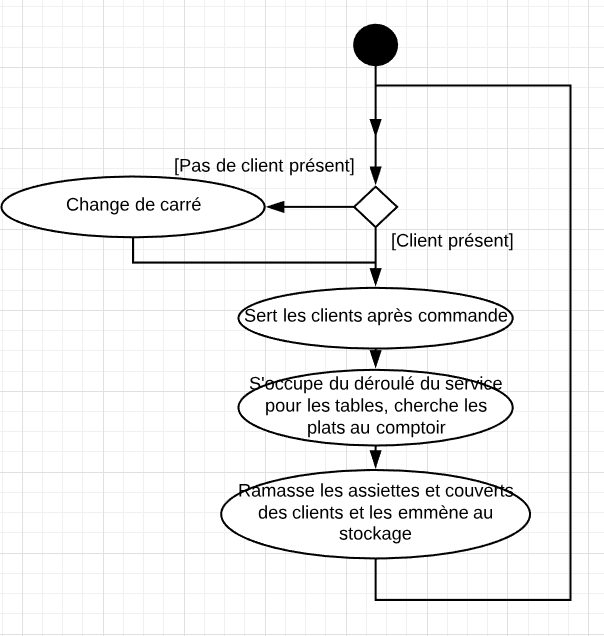
**Maitre d’hôtel =**



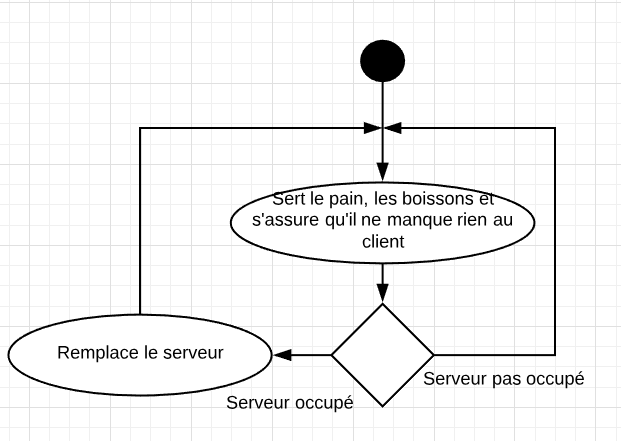
**Chef de rang =**



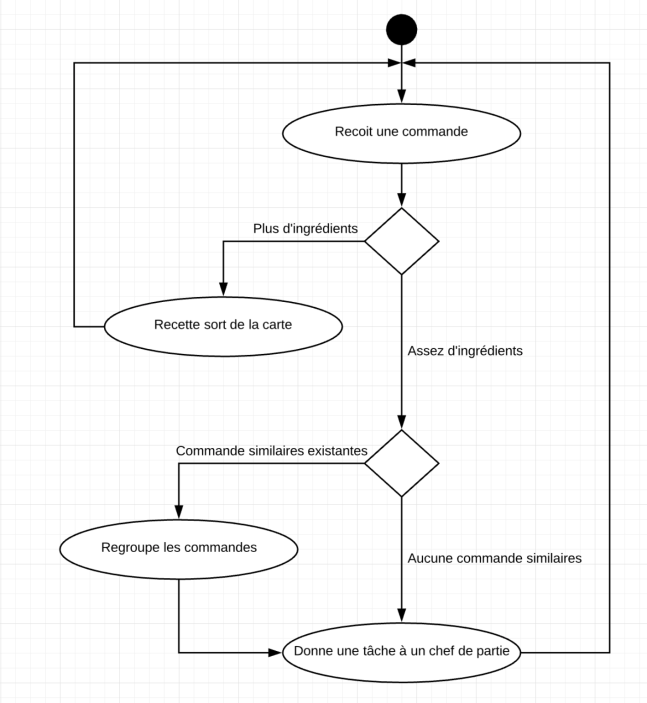
**Serveur =**

****

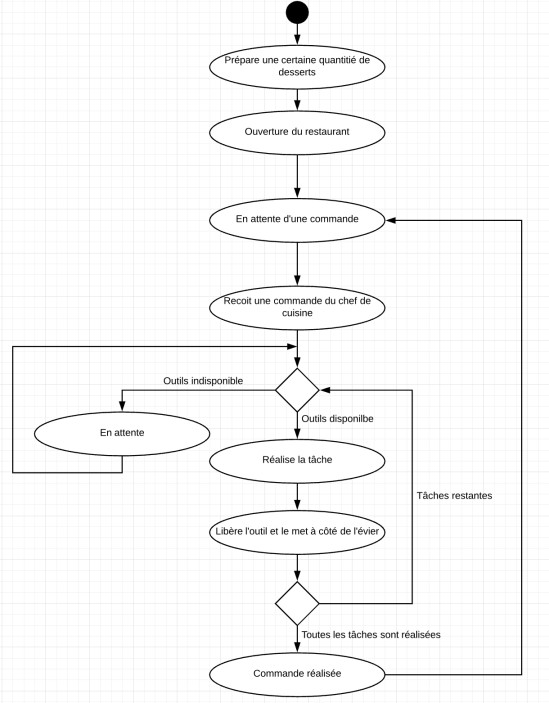
**Commis salle =**

****

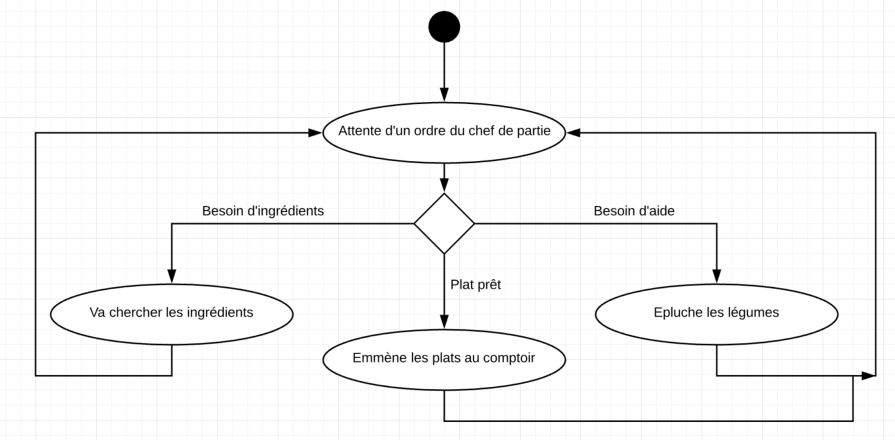
**Chef cuisinier =**

****

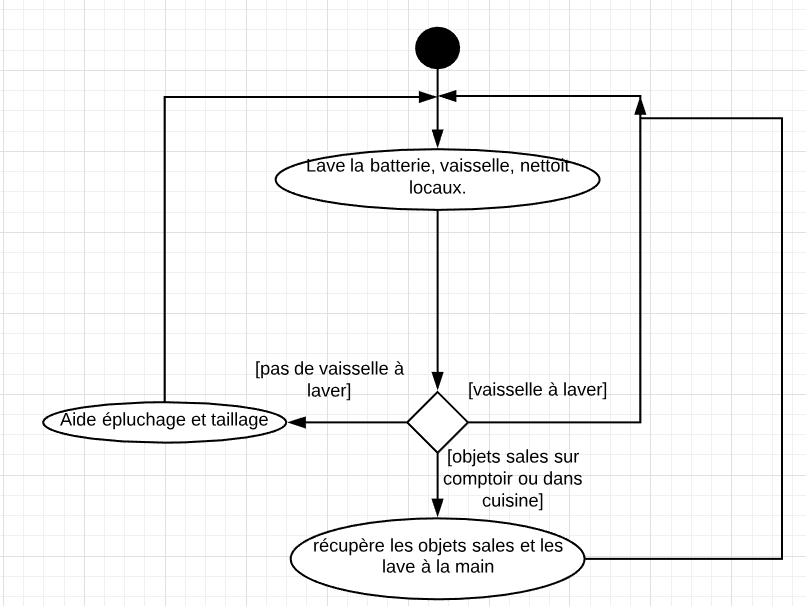
**Chef partie de cuisine =**

****

**Commis cuisine =**

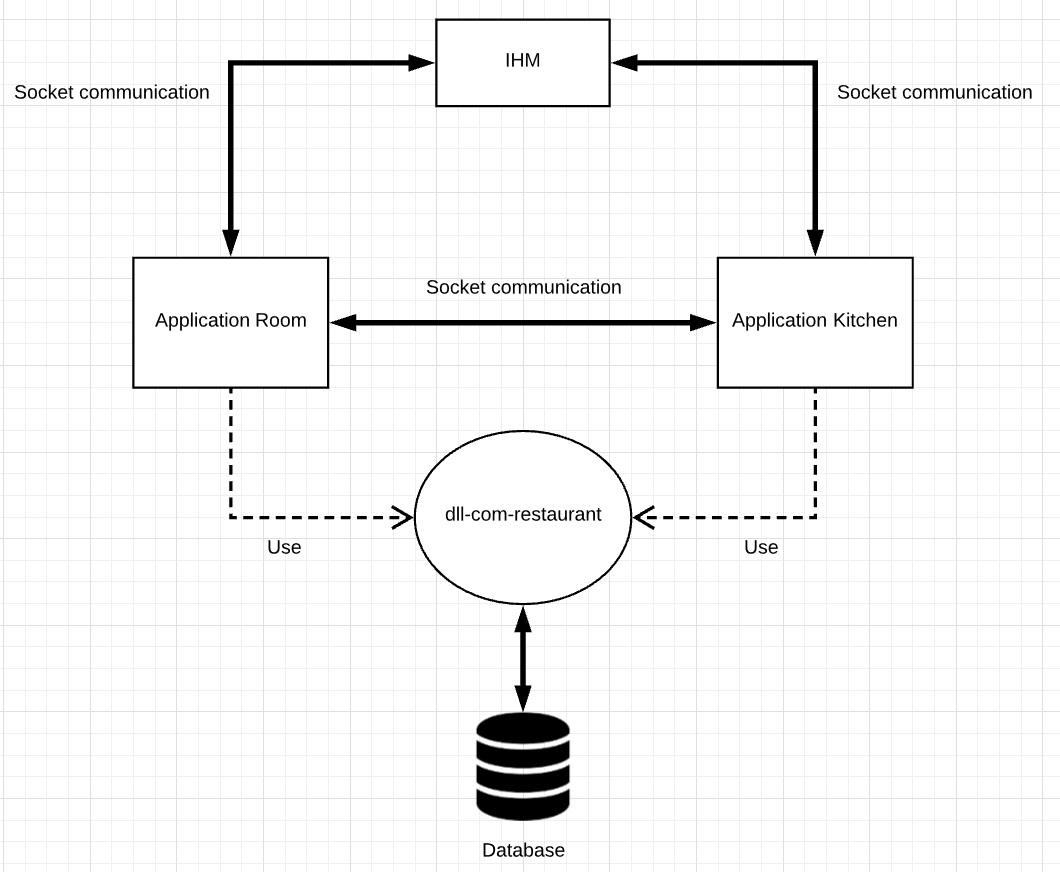
****

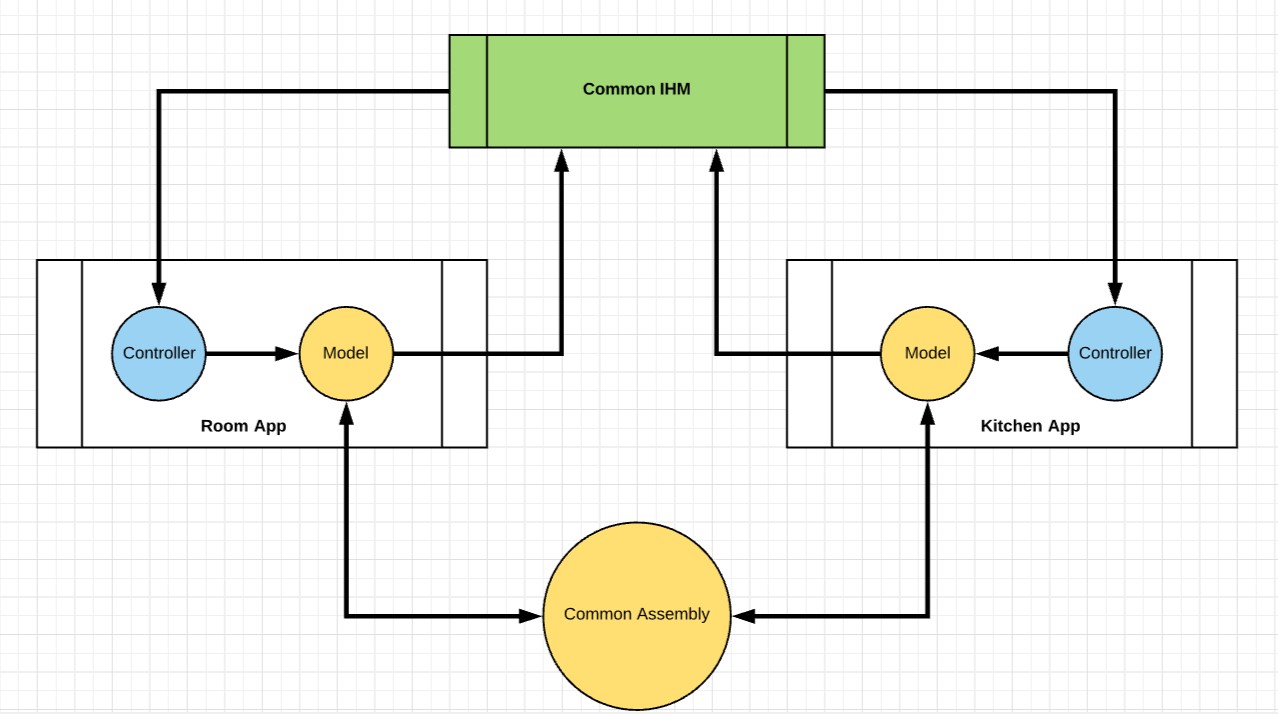
**Plongeur =**

****

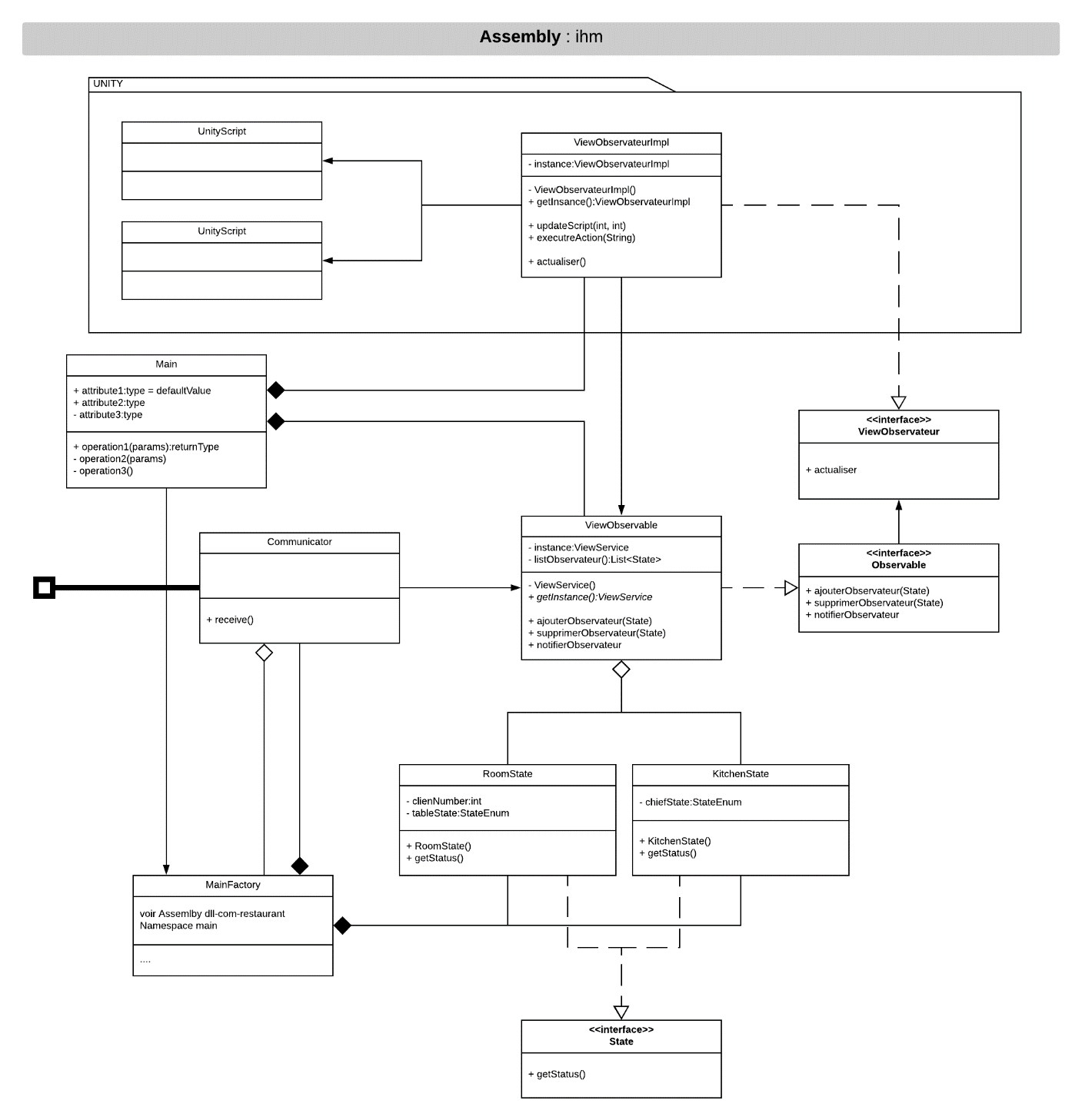
* 1. Diagrammes de composants et classes :

**Structure de notre application =**

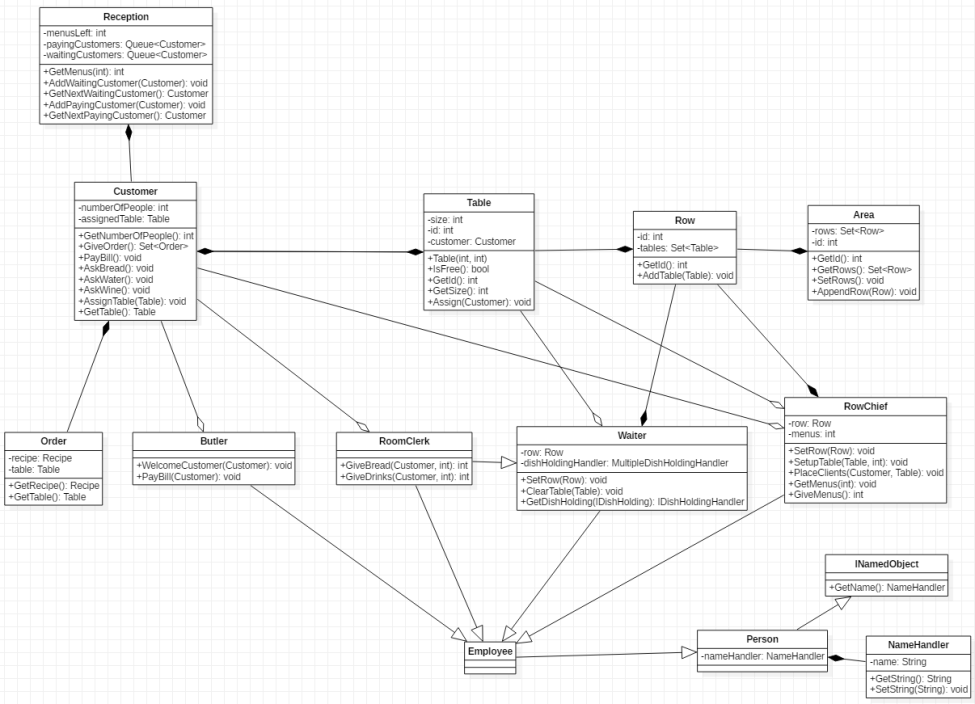
****

****

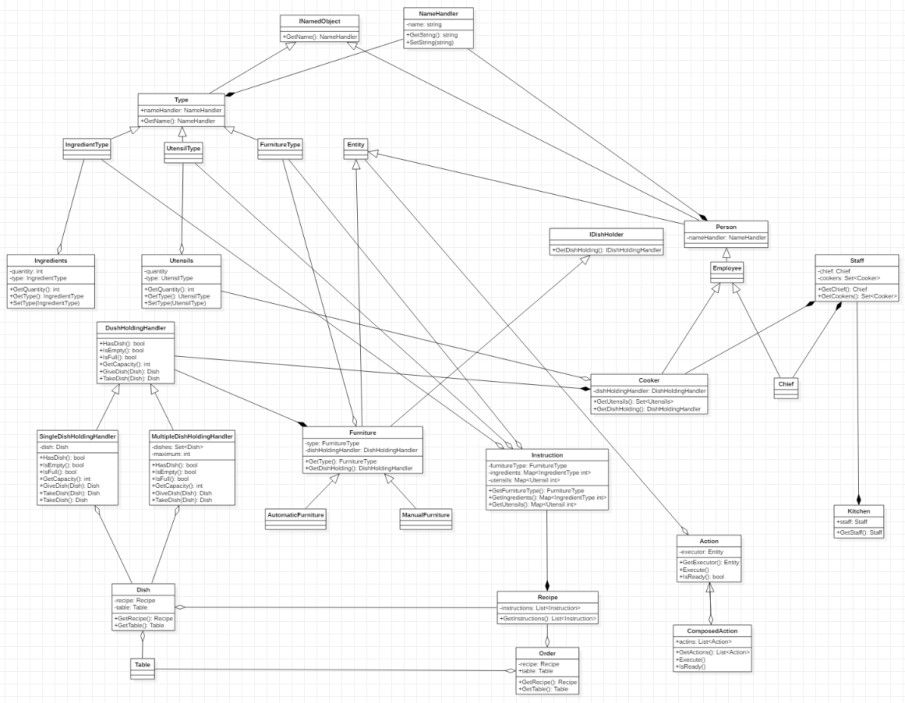
**Ihm =**

****

**Room =**

****

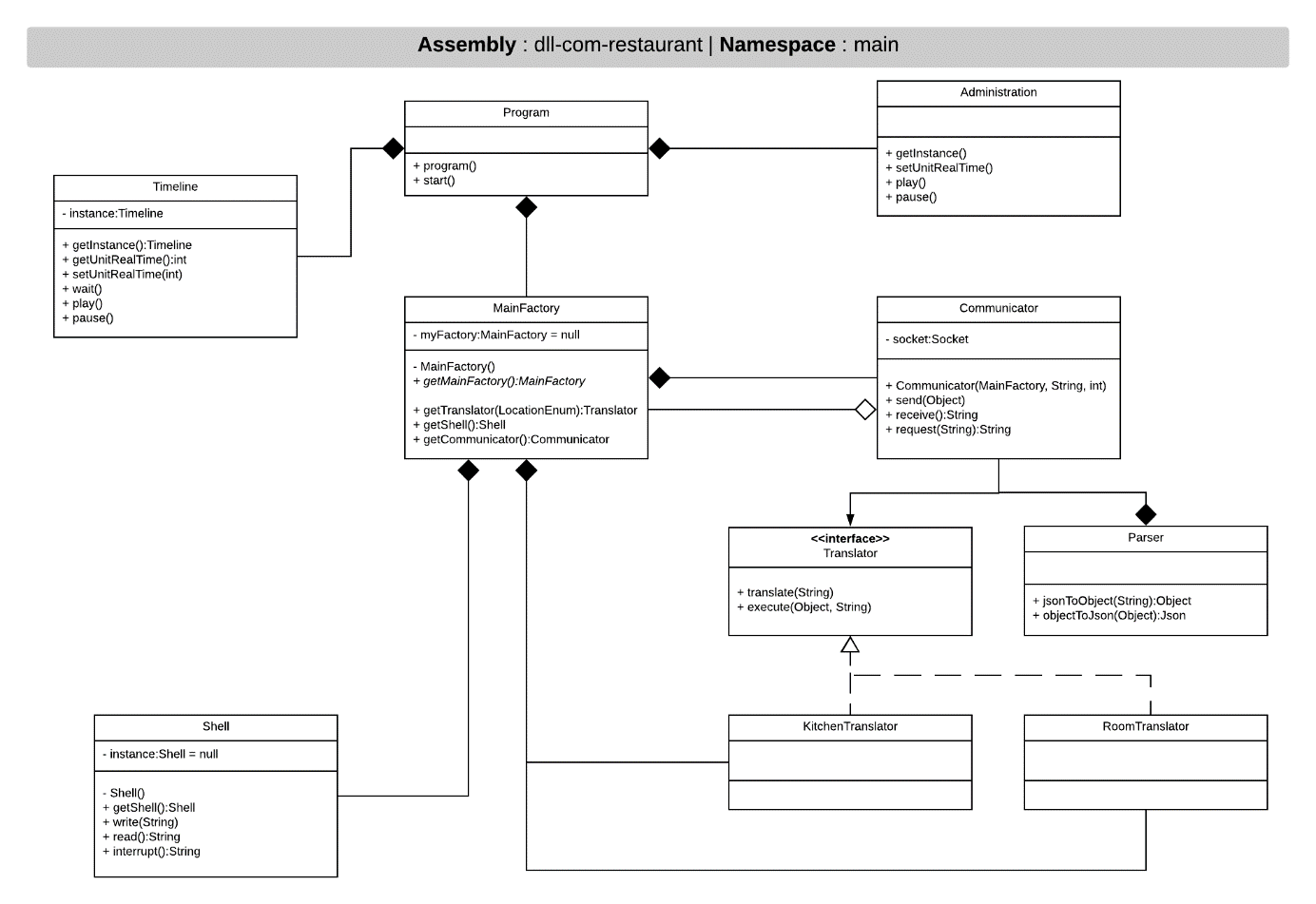
**Kitchen =**

****

**Libraries dll-com-restaurant =**

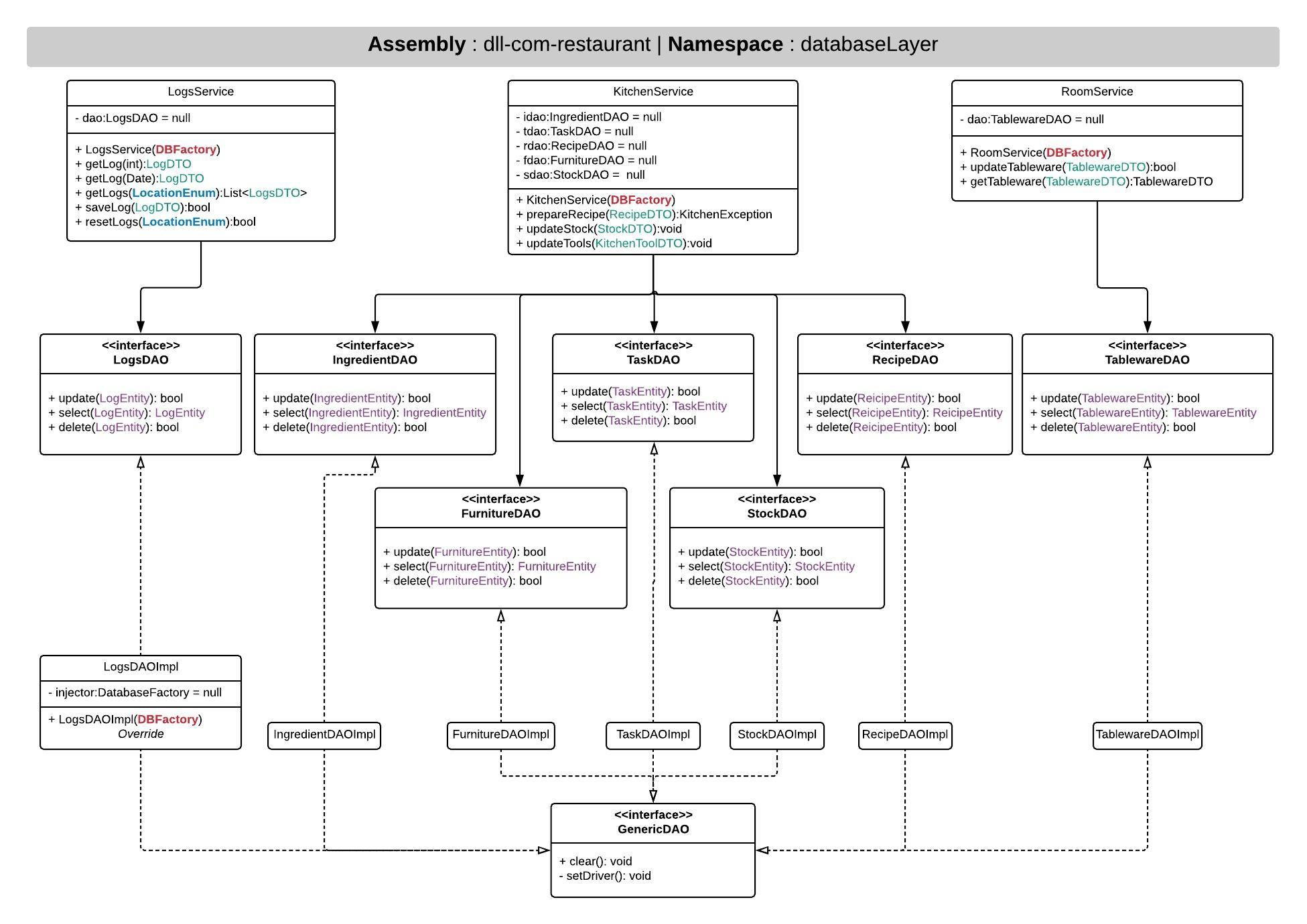
*Main :*

Le namespace Main contient l’environnement d’exécution des différentes applications ainsi que leurs moyens de communication.

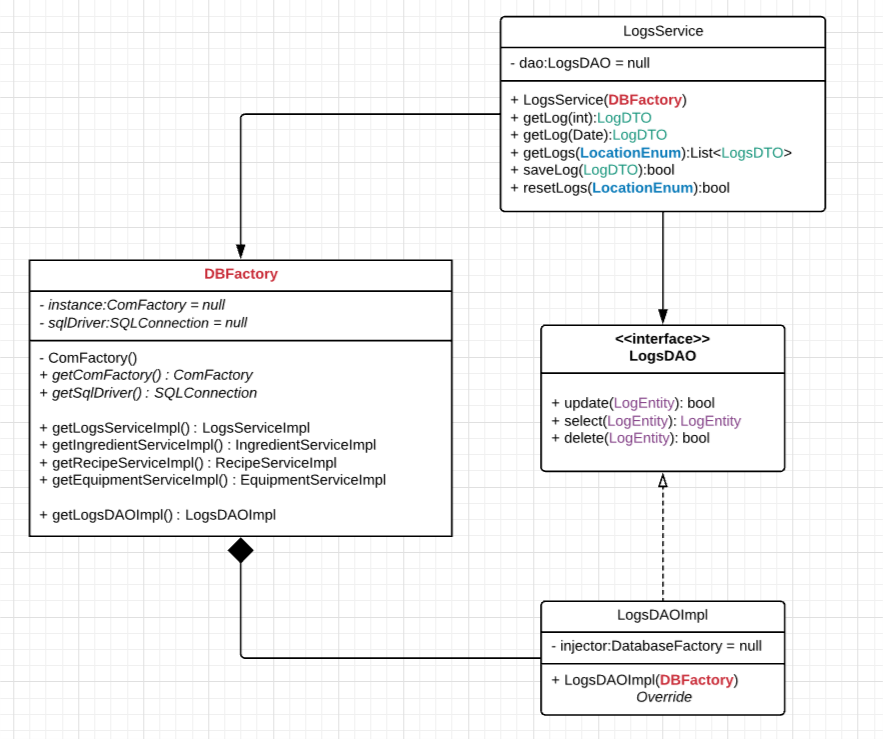
****

*Database layer :*

Les classes présentent dans le databaseLayer ont pour missions de créer la couche d’accès vers la base de données. On y retrouve le pattern DAO, les classes « Service » permettant aux applications de manipuler des données de la BDD, etc…

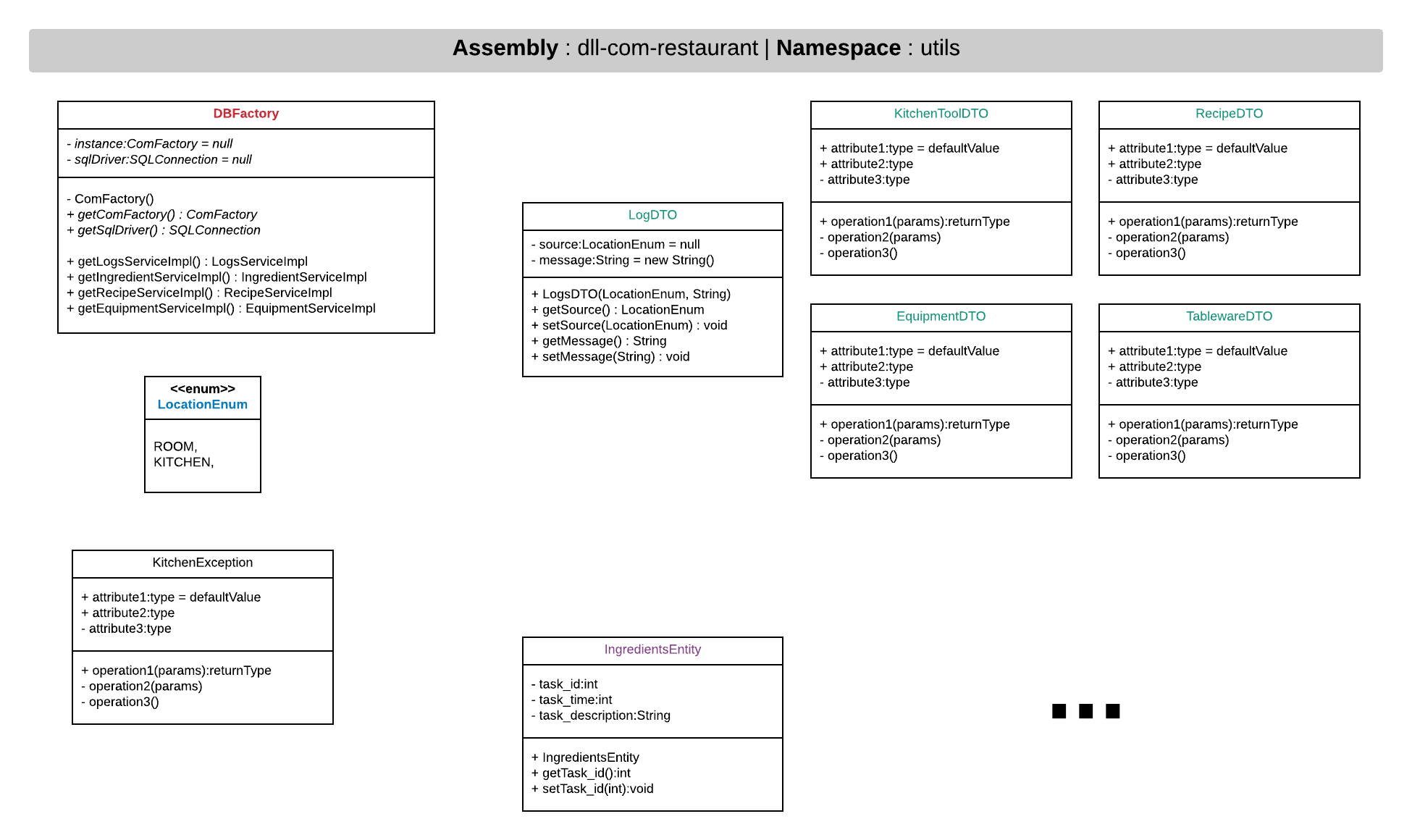


Le diagramme UML précédent concerne uniquement les classes principales et leur relation dans le namespace databaseLayer. Le schéma suivant explique plus en détails le fonctionnement d’une partie du DAO.

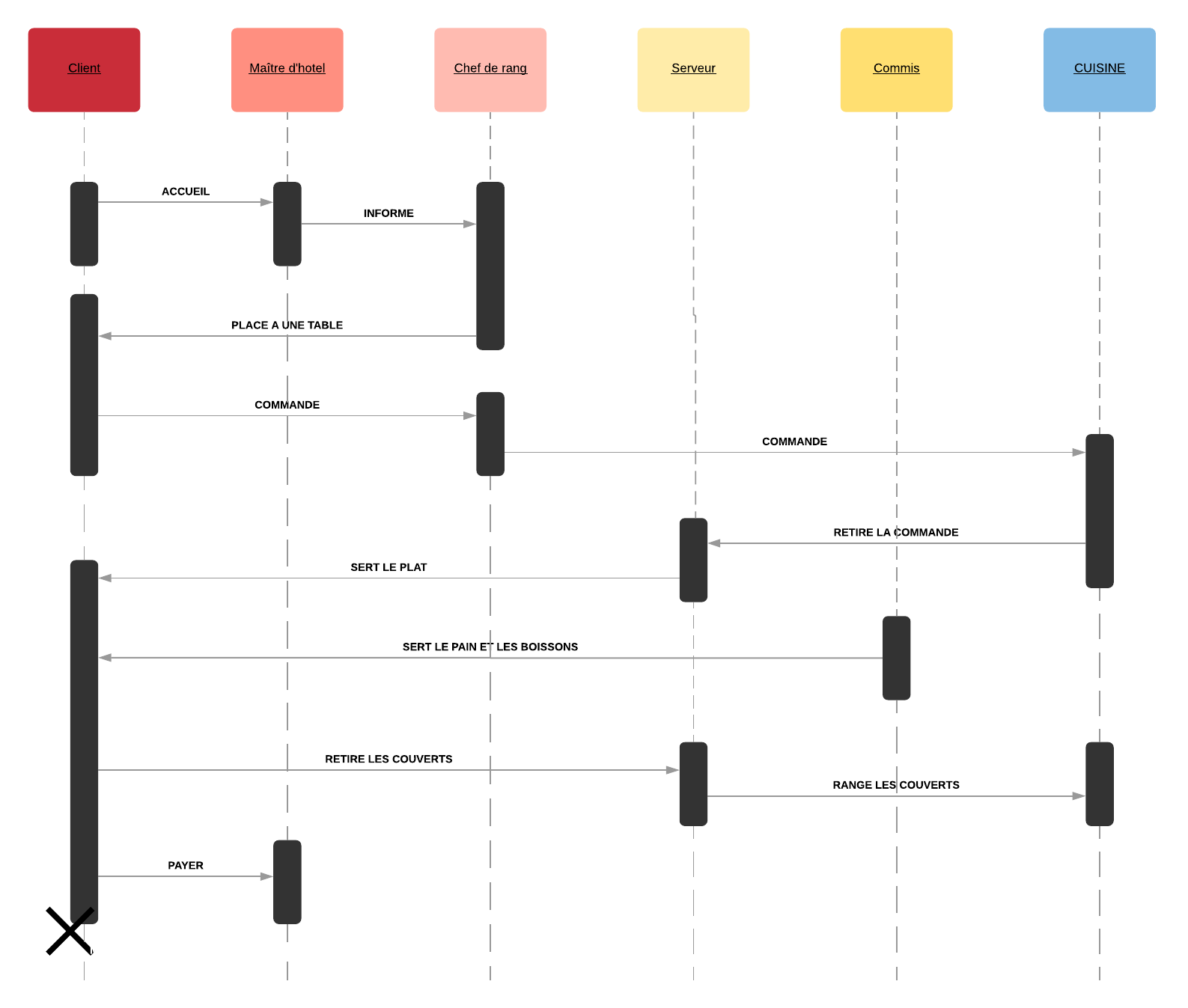


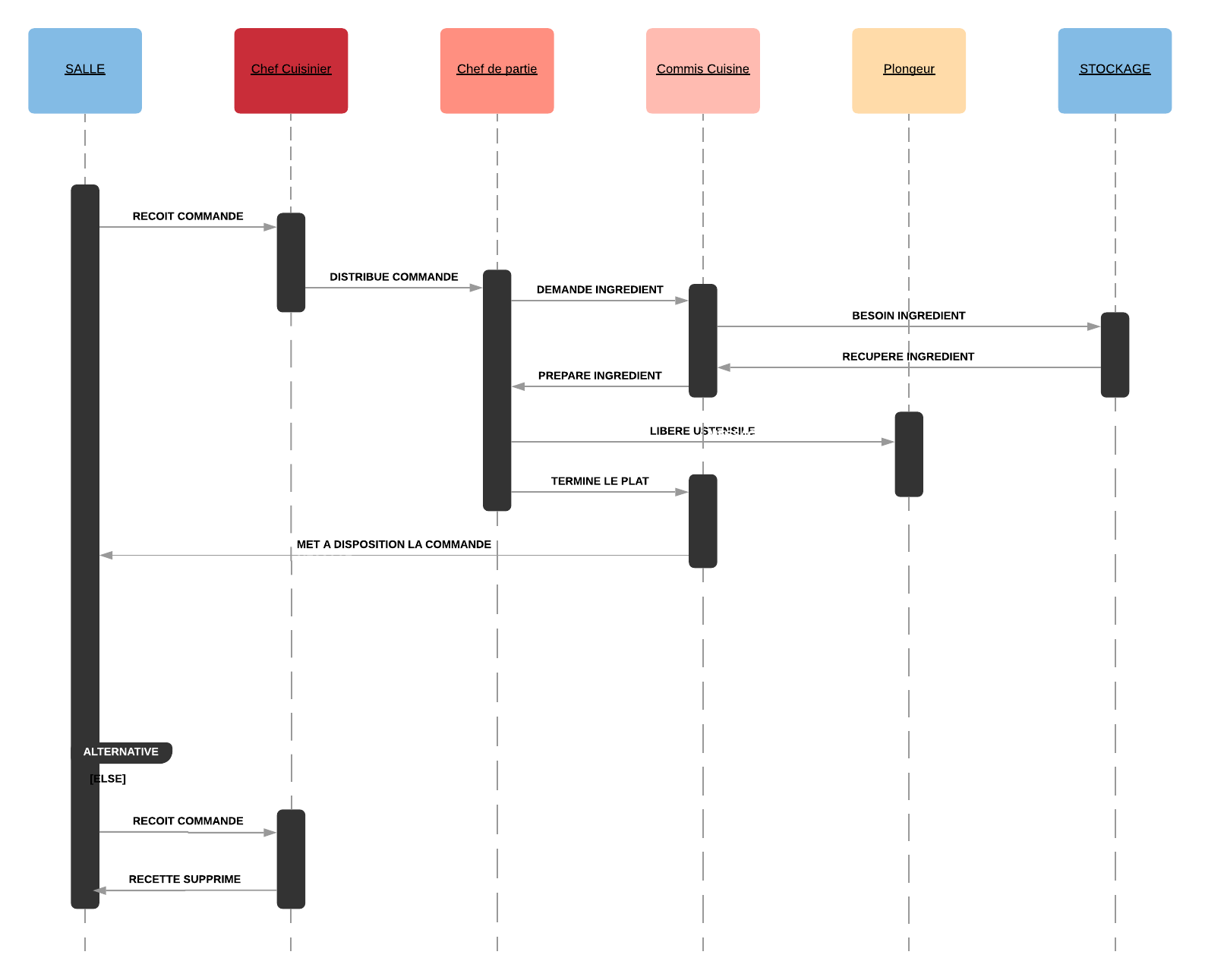
*Utils :*

Utils contient les classes de transport de la donnée, les Factory, les énums, etc… Ce namespace à pour but de fournir les outils nécessaires à la bonne exécution des classes fonctionnelles des applications Kitchen et Room.

**

* 1. Diagrammes de séquences :



**

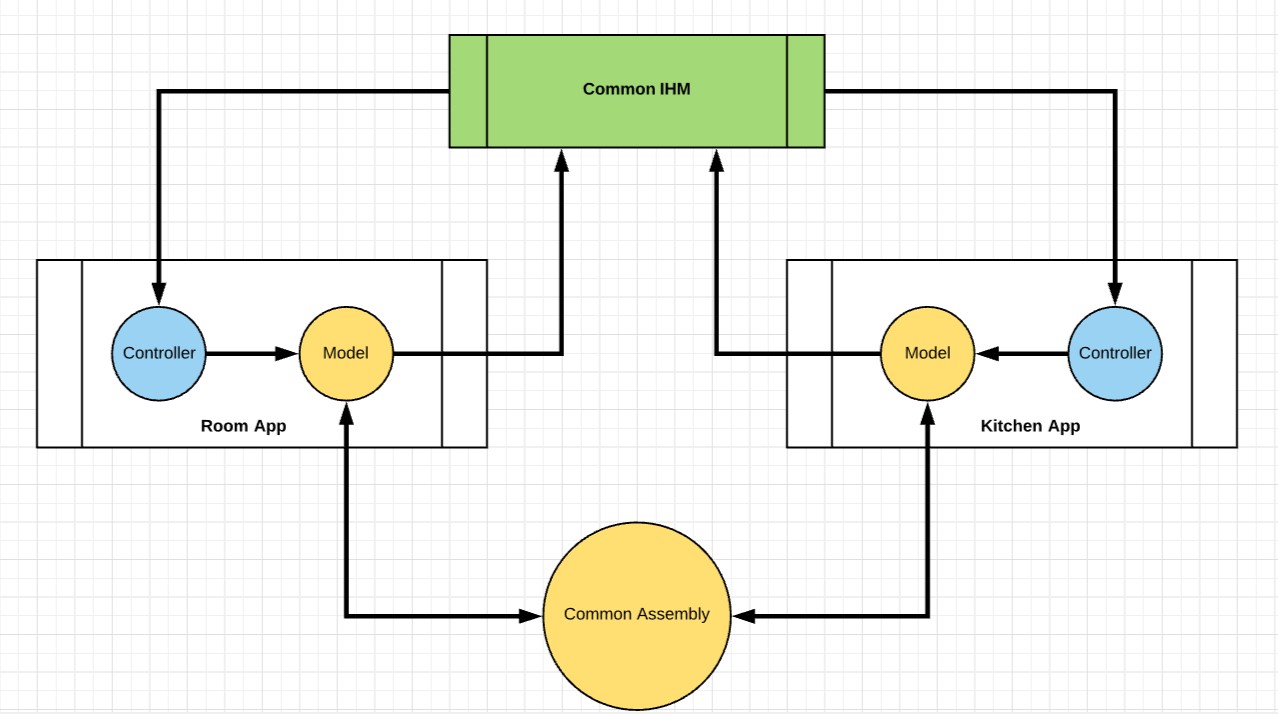
1. ***Justifications et détails des DP utilisés =***
   1. MVC :

Le pattern MVC est un motif d’architecture composé de trois grands éléments : le controller, le modèle et la vue.

La première raison d’utiliser une telle construction est la clarté qu’elle apporte : en effet, il est beaucoup plus simple de travailler en équipe sur une structure normalisé. La seconde raison est la possibilité d’évolution qu’offre une telle solution : le pattern MVC impose la séparation des grands éléments d’une application (couche métier, IHM, traitement de donnée, etc…). Cela à pour conséquence de réduire le couplage entre ces éléments et ainsi, les rendre plus facilement modifiable, remplaçable.

Par exemple, on peut très simplement modifier le code d’un élément du model comportant un calcul mathématique permettant la détermination de la position d’un personnage sur l’IHM par un autre calcul ou un moyen totalement diffèrent comme un fichier texte comportant des positions prédéfinis. Ce changement n’aura aucun impact sur notre vue : c’est la principale force du design pattern MVC.

Dans notre cas, nous avons choisi d’implémenter le pattern MVC dans nos deux applications : Room et Kitchen (voir plus haut « Structure de notre application »). La particularité est que Room et Kitchen contiennent tout deux un controller et un modèle, cependant ils partagent une vue unique.



Comme le montre le schéma plus haut, nos applications Room et Kitchen composées d’un MVC, ont en commun la vie (représenté par l’application IHM) ainsi que certains éléments du model présent dans une librairie externe (la couche d’accès à la base de données, les DTO, les classes de représentation de données, …).

* 1. Singleton :

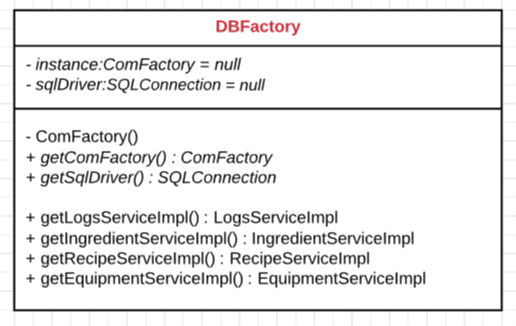
Un pattern singleton à pour objectif de restreindre l’instanciation d’une classe à un seul objet. A ne pas confondre avec le mot clé « static » qui permet uniquement d’appeler une méthode d’une classe sans passer par son instanciation.

Dans notre application, nous rencontrons plusieurs pattern singleton tel que :

* **L’utilisation de Factory** : Nos Factory sont des singletons. Cela permet d’avoir un unique distributeur d’instance et donc de permettre le contrôle de ces instances (une Factory singleton peut distribuer et controller d’autre singleton). Cette utilisation apporte donc de la lisibilité de la stabilité à notre code mais oblige la distribution de l’instance à toute l’arborescence de classe de notre application.
* **Le driver SQL**: Le driver SQL est distribué par une Factory Singleton, son instanciation est contrôlée par ce même élément ce qui permet de s’assurer qu’il y a bien un seul point d’entrée vers notre Base de données (on gagne ainsi en fiabilité et en sécurité de nos données).
  1. Factory :

Un pattern Factory permet de fournir une classe s’occupant de l’instanciation de toutes les classes de notre programme.

Comme dit plus haut, nous utilisons une Factory Singleton dans chaque programme. L’objectif est de simplifier le développement et la modification du code en rassemblant toutes les créations d’instances dans un unique endroit. Le second objectif est de réduire le couplage entre nos classes, cela simplifiera l’écriture des test unitaires ainsi que l’évolution de notre programme. Le pattern Factory est aussi un élément indispensable pour mettre en place une Injection de dépendances au niveau de la couche métier et de l’IHM pour isoler le model du reste des fonctionnalités.



Exemple Factory Singleton

* 1. Observer :

Le pattern Observateur (en anglais Observer) définit une relation entre objets de type un-à-plusieurs, de façon que, si un objet change d’état, tous ceux qui en dépendent en soient informés et mis à jour automatiquement.

On utilisera le design pattern Observer lorsque l’on souhaite qu’une classe A soit informé du changement d’un changement d’état d’une autre classe (B). On peut très bien demander à la classe B si un changement a eu lieu mais à quelle fréquence, toutes les minutes ? Toutes les secondes ? Cela surchargera d’appels la classe B.

La solution est donc de laisser la classe B informer la classe A de ses changements, elle peut très bien aussi informer plusieurs classes d’un coup (en évitant de lier fortement les classes entre elles).

Pour implémenter ce design pattern nous avons besoin de deux interfaces :

• « Observateur » : chacune des classes qui l’implémentera sera notifié lors d’un changement d’état de la classe observée

• « Observable » : chacune des classes qui l’implémentera informeras les observateurs.

Les classes Observable dispose de 4 méthodes : ajouterObservateur(Observateur), supprimerObservateur(Observateur), notifierObservateurs() et getEtat().

Dans notre cas nous utiliserons ce Design Pattern pour écouter un changement venant de Kitchen et Room et nous changerons la vue en fonction.

* 1. DAO :

Le pattern DAO impose un schéma de construction pour l’utilisation d’une base de données et donc la génération et l’utilisation de requêtes SQL. Le pattern DAO est un complément qui s’agence très facilement avec le pattern MVC. Il permet d’abstraire la base de données. Cela à pour conséquence de rendre la structure de la base de données ainsi que la façon de l’attaquer totalement transparente pour les objets métier (les objets s’occupant du traitement de donnée dans le model).

Nous utilisons le pattern DAO car il est, selon nous, indispensable de l’utiliser dans ce type de projet. Les avantages qu’il offre sont nombreux mais les plus frappants sont surement les suivants :

* Abstraction de la structure de la base de données.
* Utilisation de la base de données comme un groupement « d’objet » et non comme des entités de plusieurs tables.
* Sécurité des données.
* Souplesse en cas d’évolution du service métier.
* Le schéma UML d’un pattern DAO est tout de suite impressionnant et témoigne donc de la puissance souveraine de notre groupe.
* Mappage entre les objets métiers et les tables plus simple et rapide.
  1. DTO :

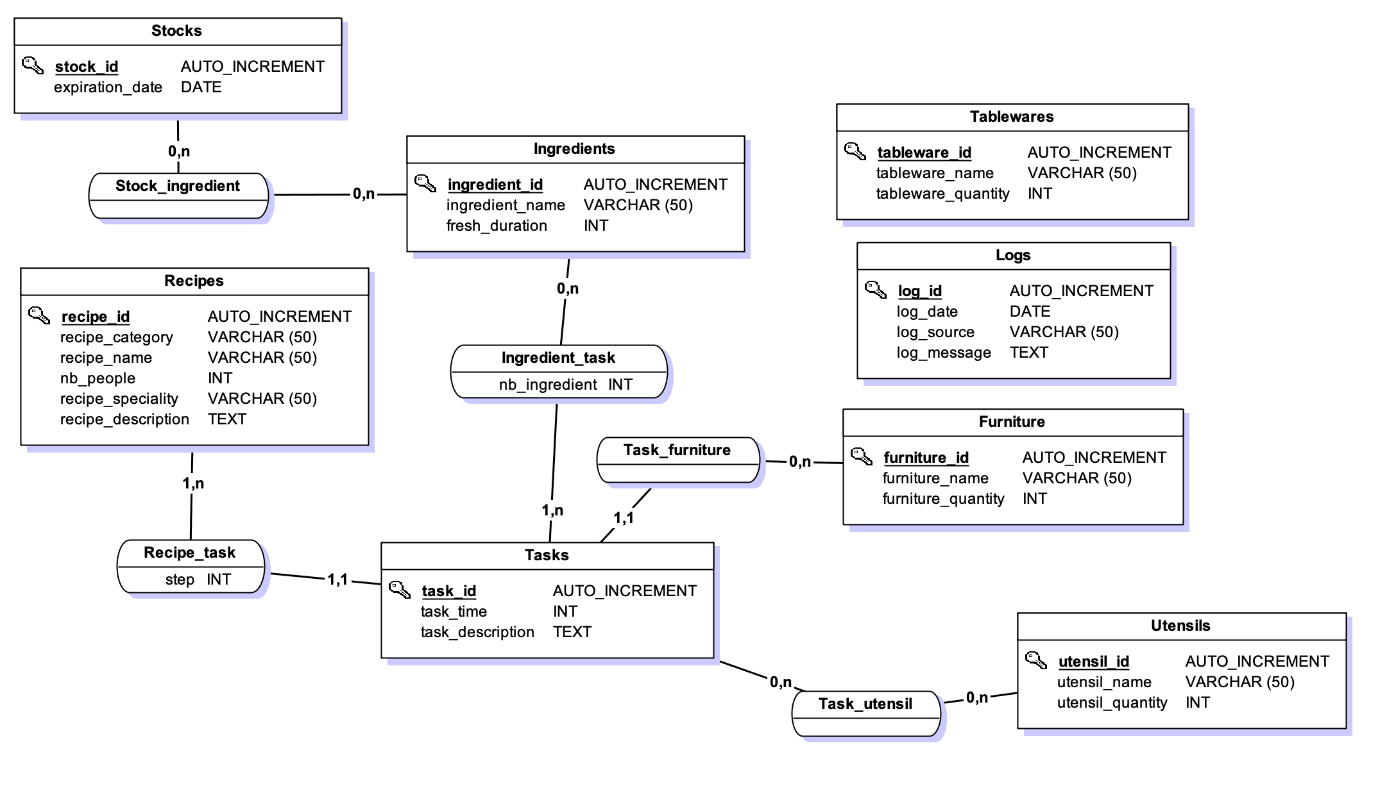
Les DTO sont des structures de données qui véhiculent l'état (les propriétés de cet objet à un instant T) des objets métiers sur le client et permettent également de remonter des informations à persister. Le principal rôle d’un DTO est donc de faire circuler l’information entre les différentes couches de notre application. Il est parfaitement adapté dans notre situation car il offre la possibilité de formaliser le transfert de données entre nos applications. Cela permettra une meilleure compréhension du code d’un membre de l’équipe par un autre.

* 1. Injection de dépendances :

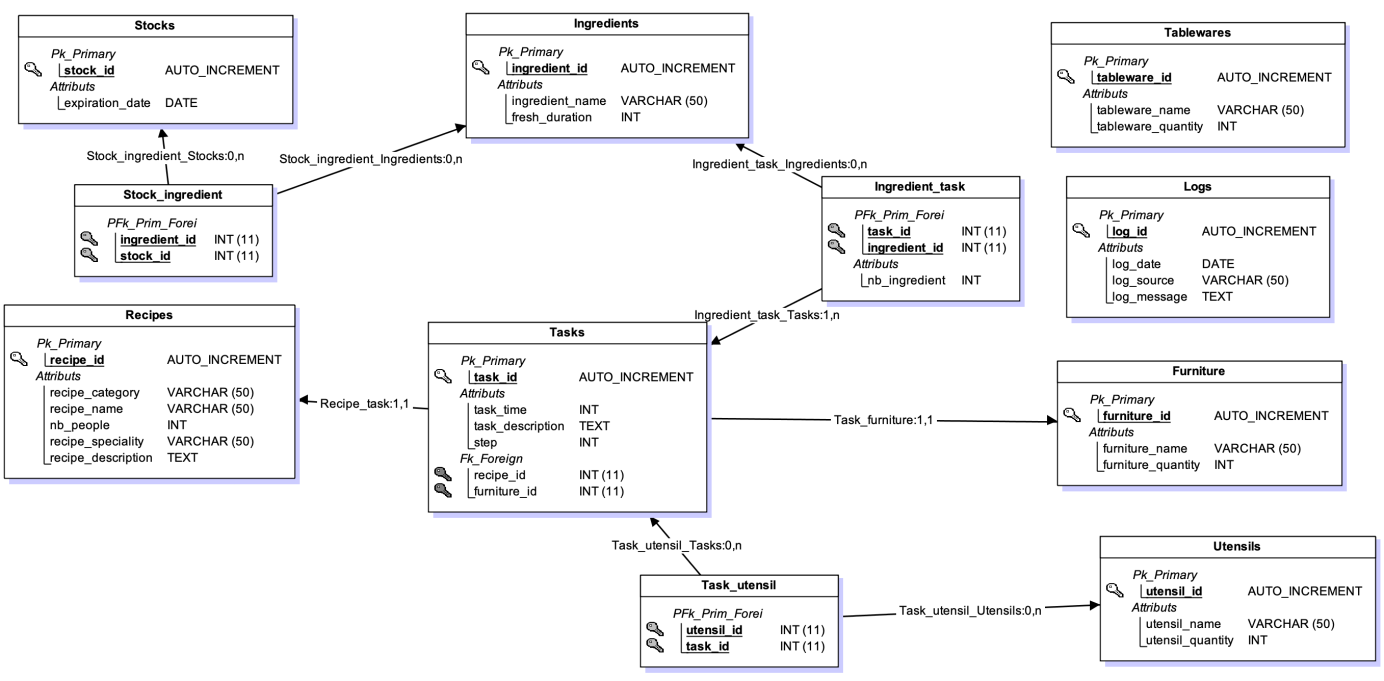
L’injection de dépendances permet de fournir un code beaucoup moins contraint par un couplage quelconque entre des classes fonctionnelles. L’application en sera plus lisible et plus facilement modifiable. Une injection de dépendances permet aussi de gérer les cas de dépendances circulaires (A utilise B et B utilise A).

Dans notre cas, l’injection de dépendances est utilisée au niveau de notre couche de données (avec les DAO notamment), cela empêche les classes fonctionnelles d’avoir conscience de l’implémentation des DAO.

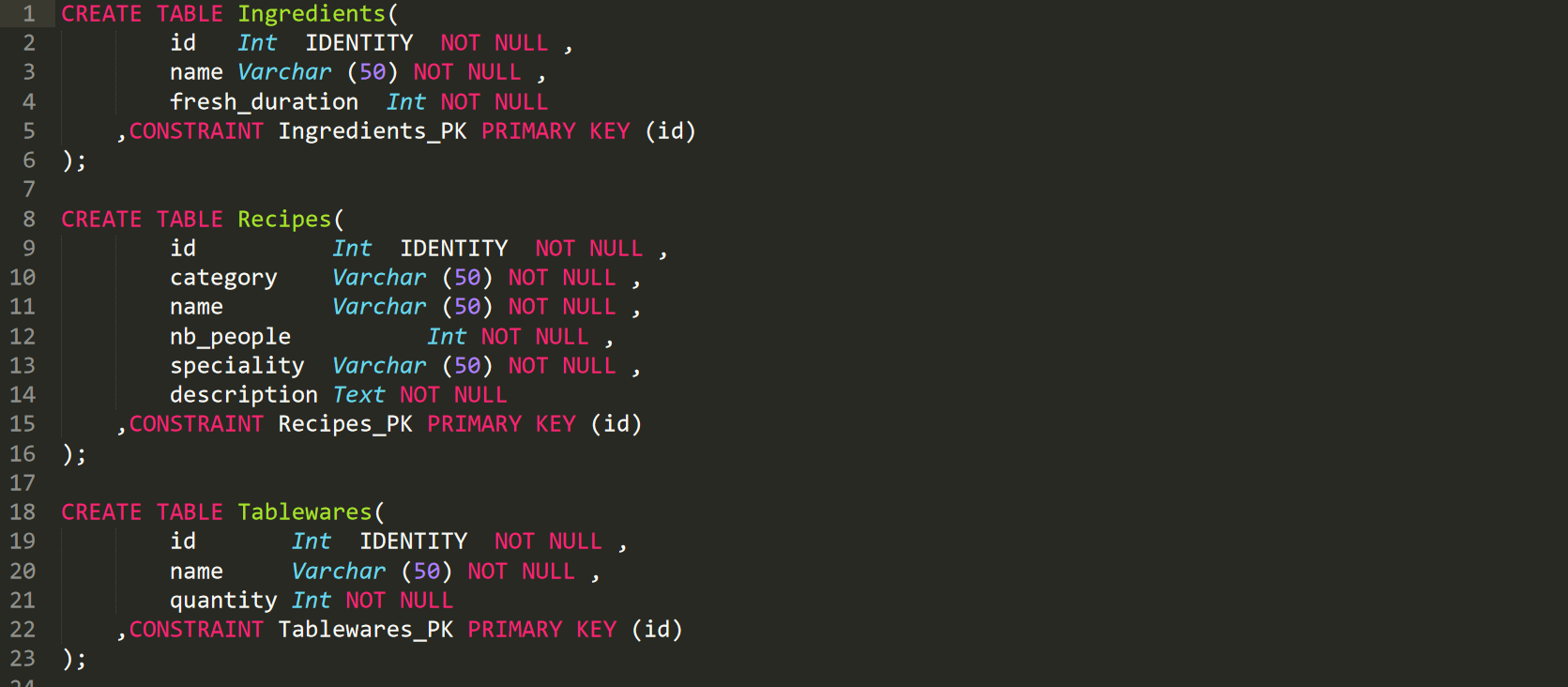
1. ***Base de données =***
   1. MCD :

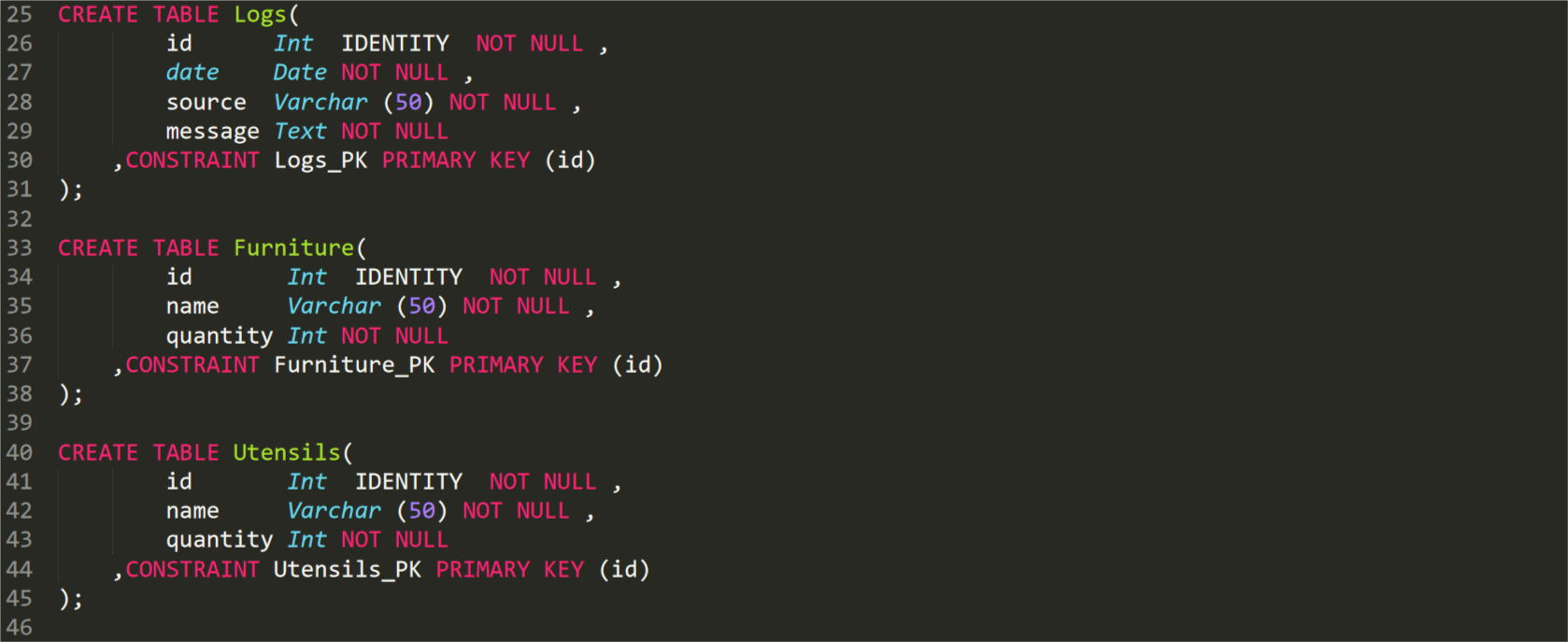


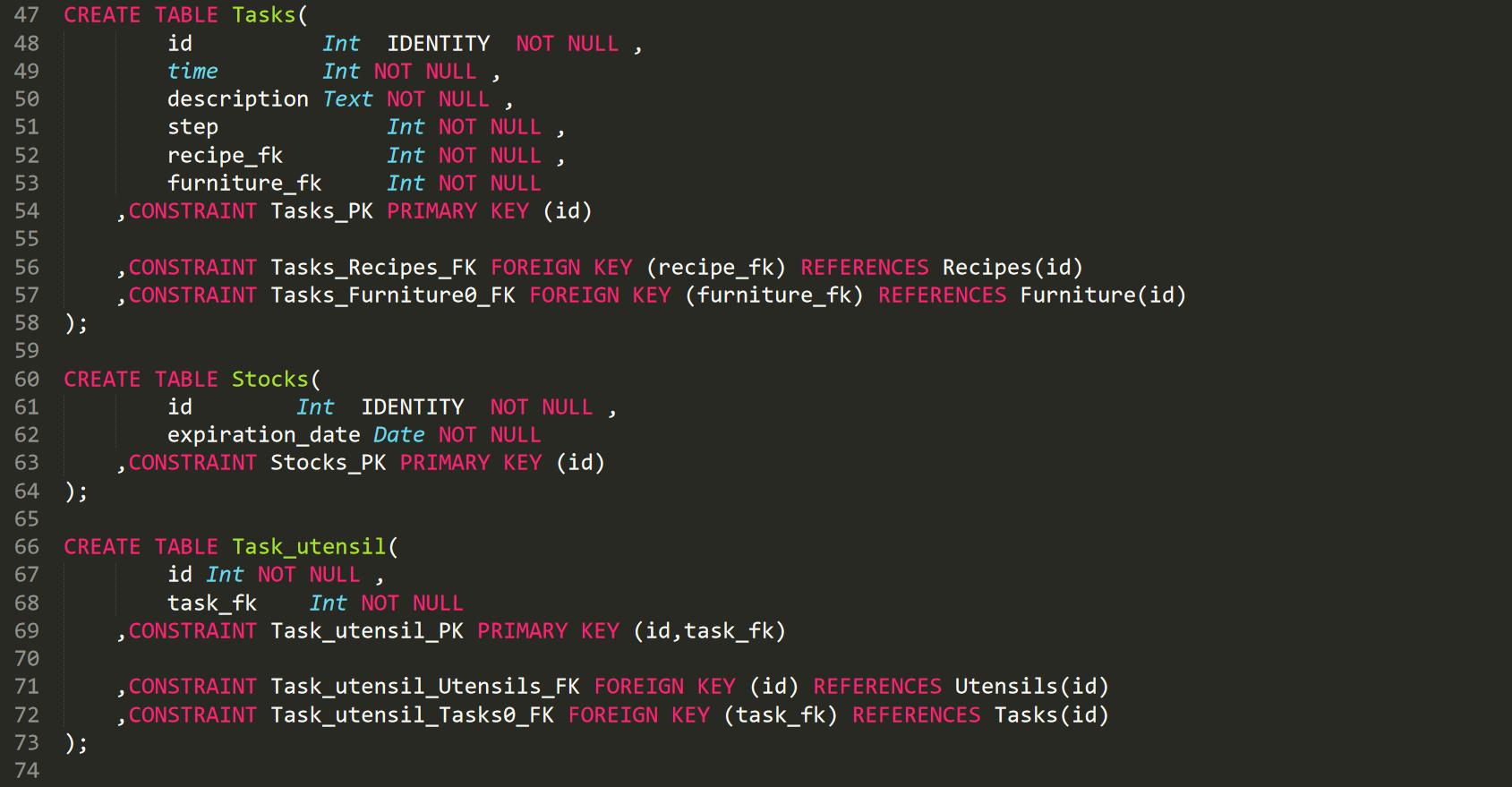
* 1. MLD :

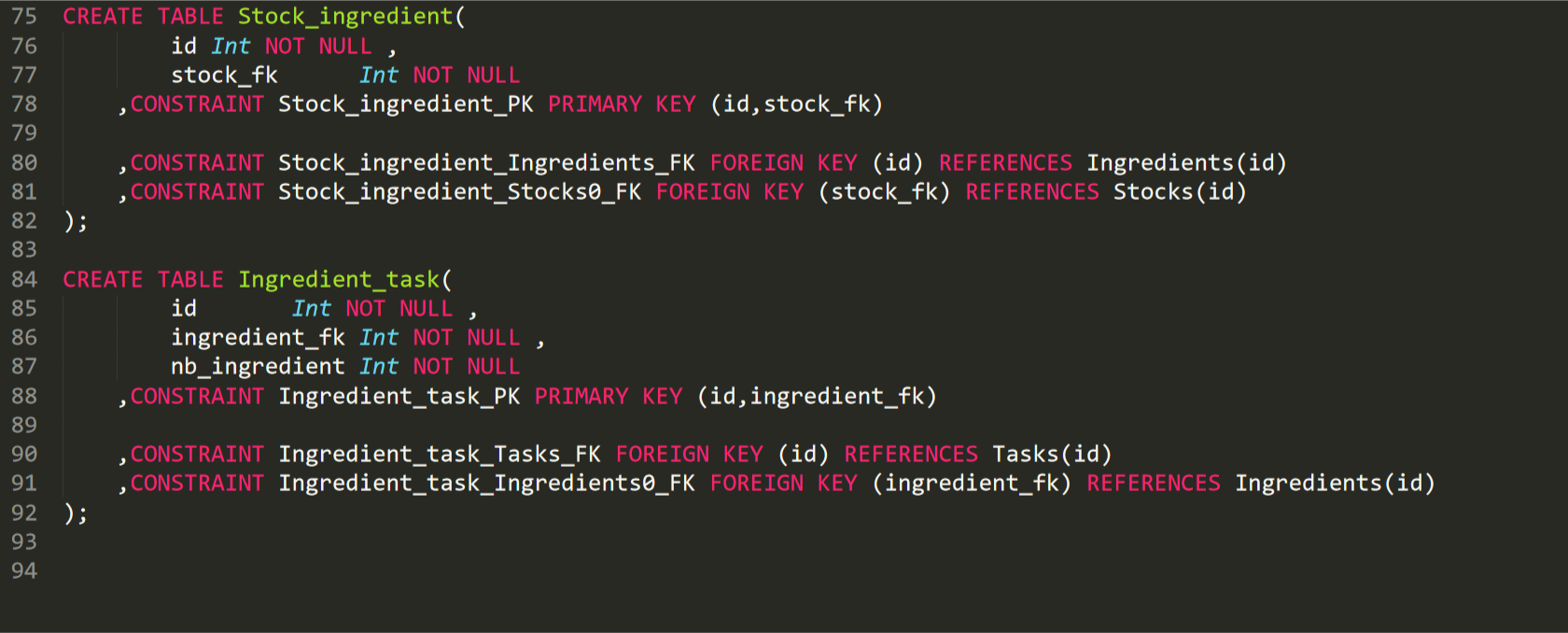


* 1. Script SQL :

**

**

**

**