Rapport d'Architecture

DING Feng

JIN Hong

WANG Yuqi

01/03/2015

TABLE DES MATIERES

1. Introduction

The *Cookie on Demand*(CoD) est un service qui a la croisée du e-Shop et du drive-in. Par la système de CoD, les clients peuvent faire leur sélection parmi le gamme des ingrédients du "The Cookie Factory" pour créer leurs cookies personnalisées en passer la commande et en spécifier la date et l'heure à laquelle ils viendront les chercher.

Dans ce projet, nous avons utilisés les connaissances de *UML* et de *Serveur Entreprise* pour construire l'architecture de cette système. Nous avons écrit les sénarios, établi la diagramme de cas d'utilisation, dressé la diagramme de classe et la diagramme de composants et la diagramme de déploiement. Alors, dans cette rapport nous allons les décrire avec minutie.

2. Specification

2.1. Sénarios

Pour réaliser cette système, il faut premièrement savoir qui pourrait utiliser cette système. En plus, les utilisateurs peuvent faire quoi par cette système. À la fin, comment réaliser les fonctionnalités. Grâce à cette logique, avant tout, nous avons écrit les quatre sénarios au-dessous.

2.1.1. Scénario 1: Création une commande

- Le client peut s'authentifier.
- Le client peut passer une commande.
- Le client authentifié peut choisir une recette pré-existante et payer par le crédit du compte quand il crée une commende.
- Le client non-authentifié ne peut pas choisir une recette pré-existante et payer par le crédit du compte quand il crée une commende.

2.1.2. Scénario 2: Opération du compte

- Le client authentifié peut gérer son compte.
- Le client non-authentifié peut créer son compte.

2.1.3. Scénario 3: Opération des informations des magasins

- Le responsable du magasin peut gérer les informations de ses propres informations.
- Le société de TCF peut administrer la liste de magasin.

2.1.4. Scénario 4: Les contraintes techniques

- Le système du CoD doit être implementé en JAVA.
- Le système d'information est développées en .Net.
    - Un système de paiement a distance.
    - Un système d'authentification des utilisateurs.
- Les service webs doivent faire l'affaire pour gérer l'interopérabilité.

2.2. Cas d'utilisation

Grâce aux scénarios au-dessus, nous avons modélisé la diagramme de cas d'utilisation, qui contient quatre acteurs et six cas d'utilisation.

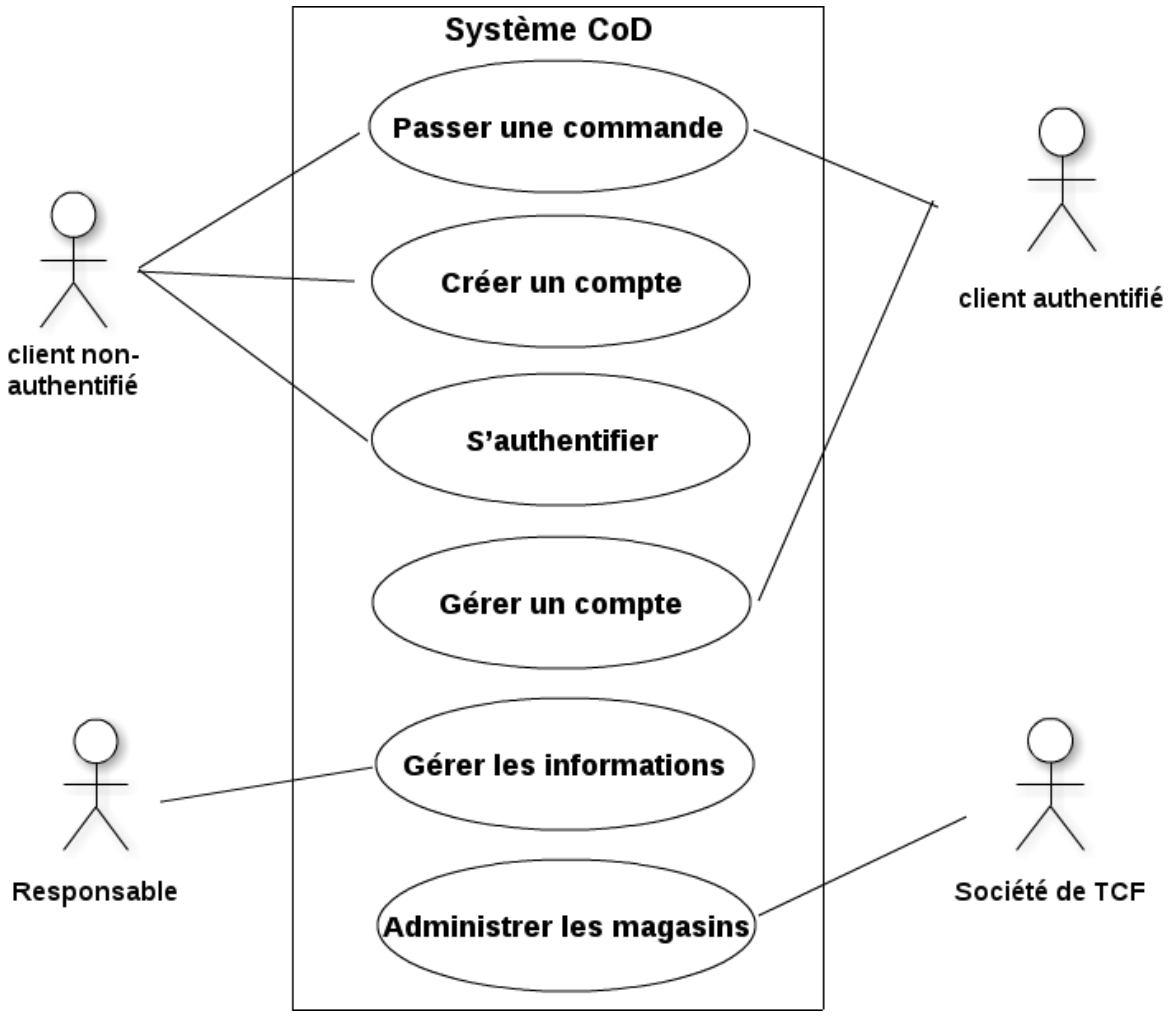


Figure 1: Diagramme de cas d'utilisation de haut niveau

2.2.1. Cas d'utilisation 1: Passer une commande

- Créer une commande
- choisir la boutique de récupération
    - choisir la date de rendez-vous
    - choisir la quantité voulue
    - choisir les recettes
        - choisir les recettes pré-éxistantes
        - créer les recettes propres(le type de pâte, les arômes optionnels, trois garnitures, le type de mélange, le type de cuisson)
- Obtenir le bon de commande
- Payer le montant de la commande
- Envoyer la commande

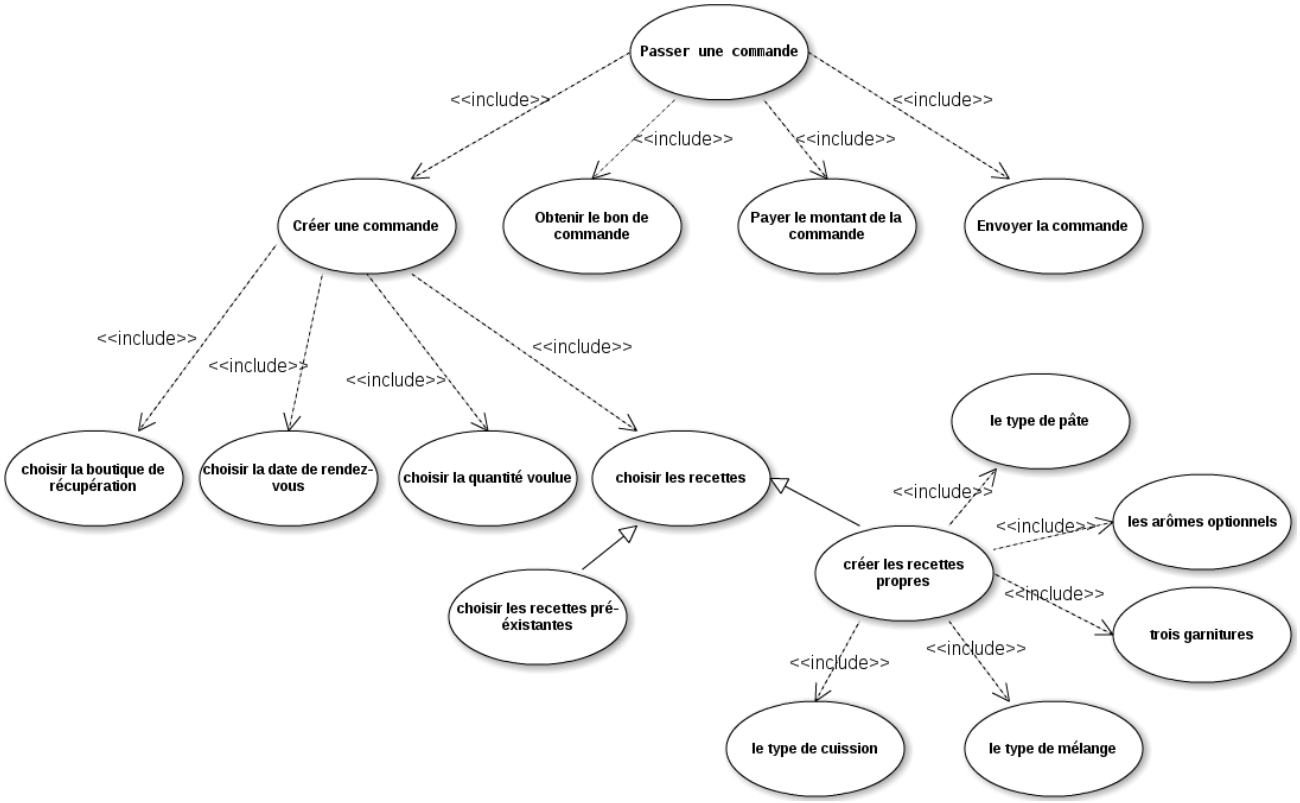


Figure 2: Diagramme de cas d'utilisation *Passer une commande*

2.2.2. Cas d'utilisation 2: S'authentifier; Cas d'utilisation 3: Créer un compte

Si l'utilisateur voulait avoir plus de pouvoirs, il pourrait premièrement s'authentifier et ensuite créer un compte.

2.2.3. Cas d'utilisation 4: Gérer un compte

- Enregistrer des préférences
  - Enregistrer les recettes personnalisées et les renommer
  - Enregistrer les lieux de récupération préférées
  - Enregistrer les informations de paiement
- Faire des propositions
- Créditer le compte par la carte cadeau

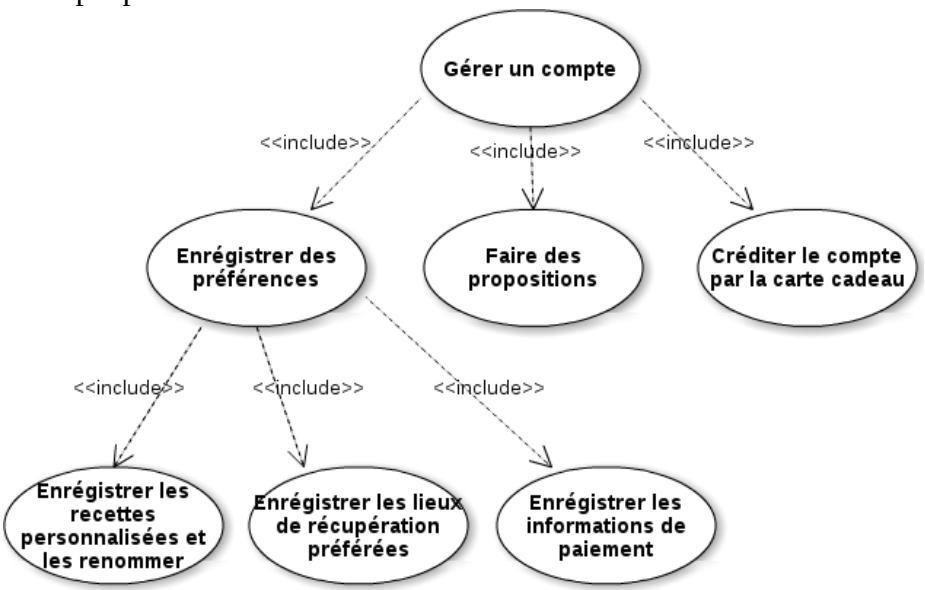


Figure 3: Diagramme de cas d'utilisation *Gérer un compte*

2.2.4. Cas d'utilisation 5: Gérer les informations

- Proposer une nouvelle recette tous les mois
- Définir les horaires d'ouverture et modifier la recette "Today's special"
- Enregistrer des statistiques d'utilisation de CoD
  - Enrégistrer l'heure de récupération
  - Enregistrer les recettes préférées des clients
  - Noter les chiffres de vente (journée, semaine, mois et année)
- Configurer les taxes

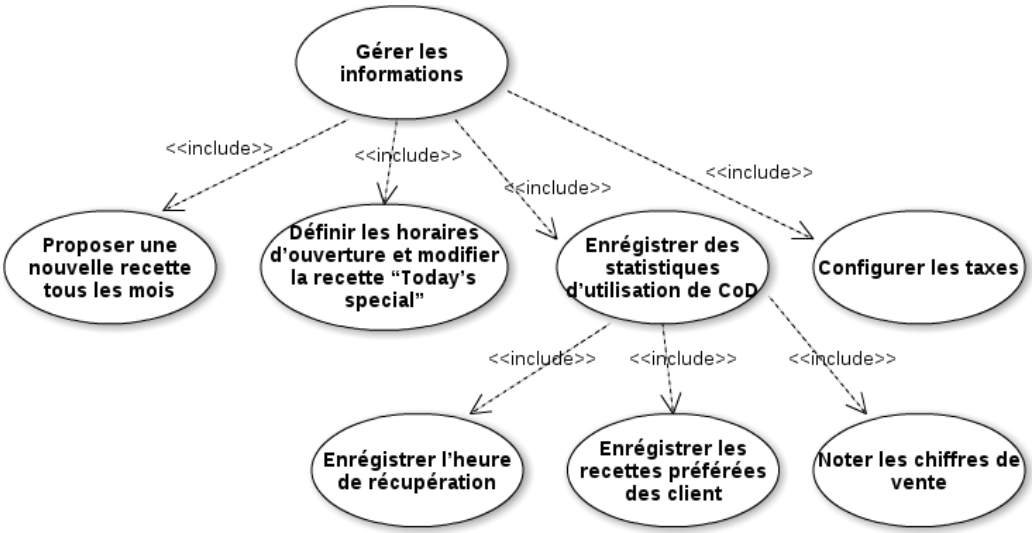


Figure 4: Diagramme de cas d'utilisation *Gérer les informations*

2.2.5. Cas d'utilisation 6: Administrer les magasins

- Ajouter un magasin
- Supprimer un magasin

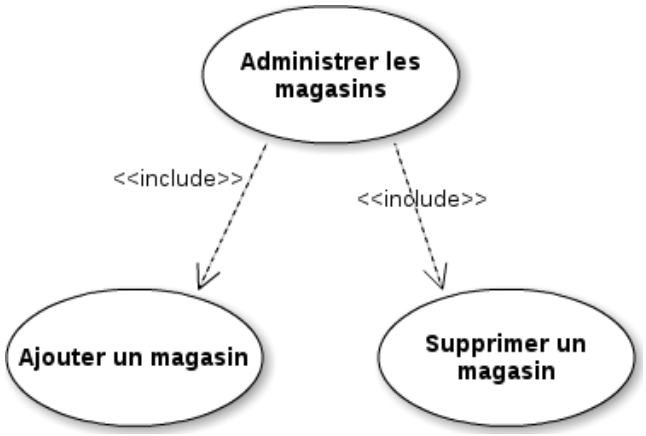


Figure 5: Diagramme de cas d'utilisation *Administrer les magasins*

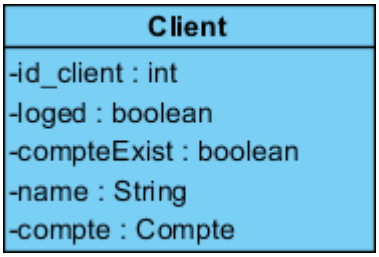
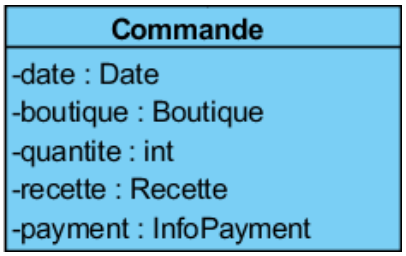
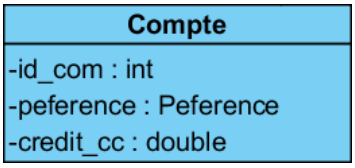
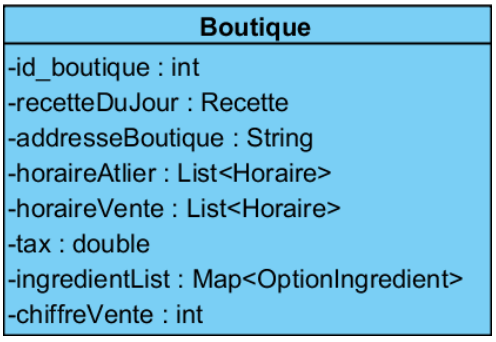
3. Architecture

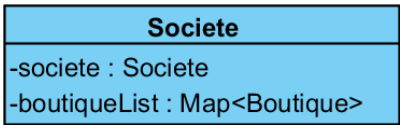
3.1. Diagramme de classe

Pour construire un diagramme de classes, nous voulons présenter premièrement notre idée et ensuite expiquer pourquoi nous l'avons effectué comme celui-là

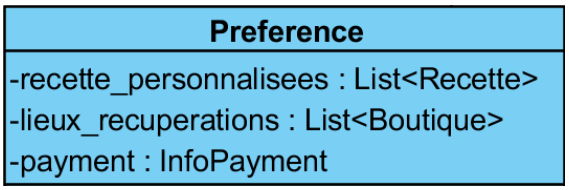
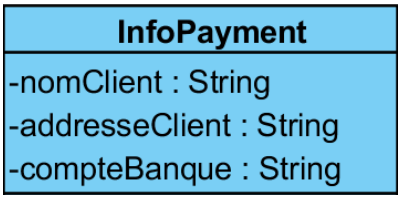
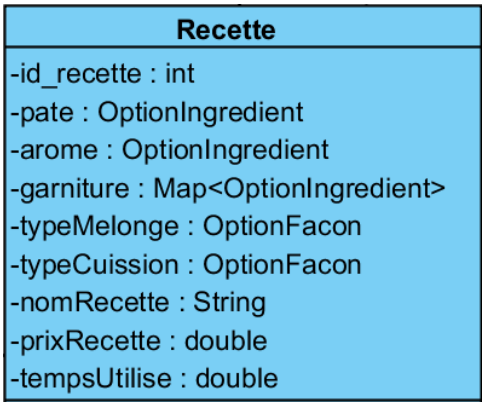
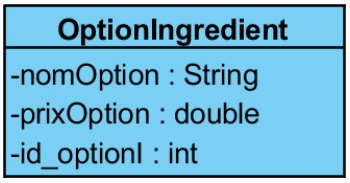
3.1.1. Des Classes

Grâce à notre diagramme de cas d'utilisation, nous avons tout d'abord établi les classes les plus importantes du système: la classe ***Client***, la classe ***Commande***, la classe ***Compte***, la classe ***Boutique*** et la classe ***Société***.

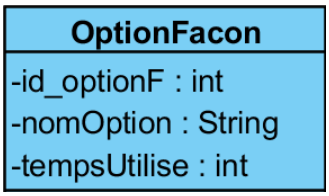
	La classe ***Client*** qui représente l'ensemble de tous les utilisateurs pouvant faire les commandes. Elle contient le nom d'utilisateur (*name*), nous avons aussi ajouté les attributes *loged* et *compteExist* qui peuvent vérifier est-ce que l'utilisateur s'est authentifié et est-ce que l'utilisateur a un compte.
	La classe ***Commande*** représente les commandes étant créées par les utilisateurs. Elle a beaucoup d'informations, la date de création (*date*), la boutique de récupération (*boutique*), le quantité (*quantite*), la recette que l'utilisateur a sélectioné (*recette*) et les informations de paiement (*payment*).
	La classe ***Compte*** contient les informations personnalisées des utilisateurs. L'attribute *credit_cc* est la crédit qui est crédité par la carte cadeau dans le compte.
	La classe **Boutique** représente toutes les boutiques. Chaque boutique a sa recette du jour (*receetteDuJour*), son addresse (*addresseBoutique*), son horaire (*horaireAtlier, horaireVente*), sa taxe (*tax*), des ingrédients existants (*ingredientList*) et des chiffres de vente (*chiffreVente*).

	La classe ***Société*** peut ajouter et supprimer les boutiques.

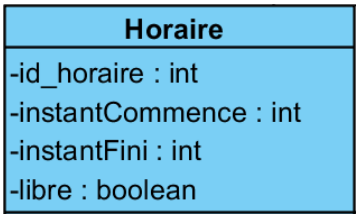
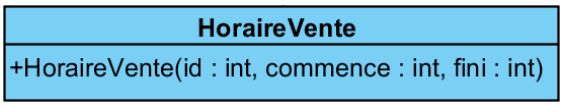
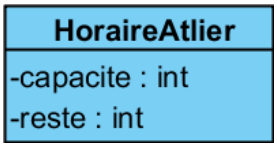
Ensuite, puisque les utilisateur ayant un compte peuvent enregistrer leur préférences qui contiennent, par exemple, les recettes personnalisées et les informations de paiement, chaque boutique a sa recette du jour et chaque recette a son type de pâte, son arôme optionnel et son type de mélange, etc. Si toutes les informations sont dans la même classe, la classe sera très lourd. Alors, pour réduire la taille des classes, et améliorer l'architecture et la portabilité de chaque classe, nous avons établi la classe ***Préférence***, la classe ***InfoPayment*** qui représente toutes les informations d'utilisateur, la classe ***Recette***[1], la classe ***OptionIngredient*** qui représente tous les ingrédients étant dans un boutique ou dans une recette et la classe ***OptionFaçon*** représentant les types de cuisson et les types de mélange.

	La classe ***Préférence*** représente la préférence d'utilisateur qui est enregistrée dans son compte. Elle contient les recettes personnalisées (*recette_personnalisees*), les lieux de récupération préférées (*lieux_recuperations*) et les informations de paiement (*payment*).
	La classe ***InfoPayment*** représante les informations qui sont enregistrées par l'utilisateur dans son compte. Par exemple, leur noms (nomClient), leur addresses (addresseClient) et leur comptes bancaires (compteBanque).
	La classe ***Recette*** représente l'ensemble de recettes qui sont créées par tous les utilisateurs. Elle contient le type de pâte (*pate*), l'arôme optionel (*arome*), les garnitures (*garniture*), le type de mélange (*typeMelange*), le type de cuisson (*typeCuisson*), le nom de la recette (*nomRecette*), le prix de la recette (*prixRecette*) et le temps utilisé pour la faire (*tempsUtilise*).
	La classe ***OptionIngredient*** représente l'ensemble d'ingrédient. Nous l'avons créé parce qu'elle nous aide pour calculer le prix d'une recette.

[1] Pour passer une commande, le client doit choisir une recette. Les recettes pré-existantes sont dans la base de donnée.

	La classe ***OptionFacon*** représente l'ensemble de type de cuisson et de mélange. Nous l'avons créé parce qu'elle nous aide pour calculer le temps utilisé pour faire une recette. Alors, nous pouvons organiser les horaires de chaque boutique.

À la fin, chaque boutique a son horaire d'atlier et de vente pour s'organiser les plannings, donc nous avons la classe ***Horaire*** qui a deux sous-classe: la classe ***HoraireAtlier*** et la classe ***HoraireVente***.

	Nous avons cr ééla classe ***Horaire*** pour permettre les boutiques s'organiser leur propres horaires. Nous s éparons un jour à quelques interval de temps. Nous pouvons le d éfinir est-ce qu'il est libre ou pas.
	La classe ***HoraireVente*** h érite la classe *Horaire*.
	La classe ***HoraireAtlier*** h érite la classe *Horaire*, mais elle a ses sp écialités. Dans un moment, l'atelier a sa capacit émaximale (*capacite*) pour fabriquer des cookies. S'il n'y a pas de place libre en ce moment, la boutique doit placer la fabriquation dans un autre moment.

Nous avons aussi établi les classes ***Gateway*** et les classes ***Finder***.
Les classes *Finder* servent à lire la base de donn ées et les classes *Gateway* lui apportent des modification.

3.1.2. Des relations

- **Relation de *Client-compte***

Grâce au sujet nous savons qu'il est possible pour n'importe qui de créer une commande, c'est-à-dire, un client ne faut pas obligatoirement avoir un compte.

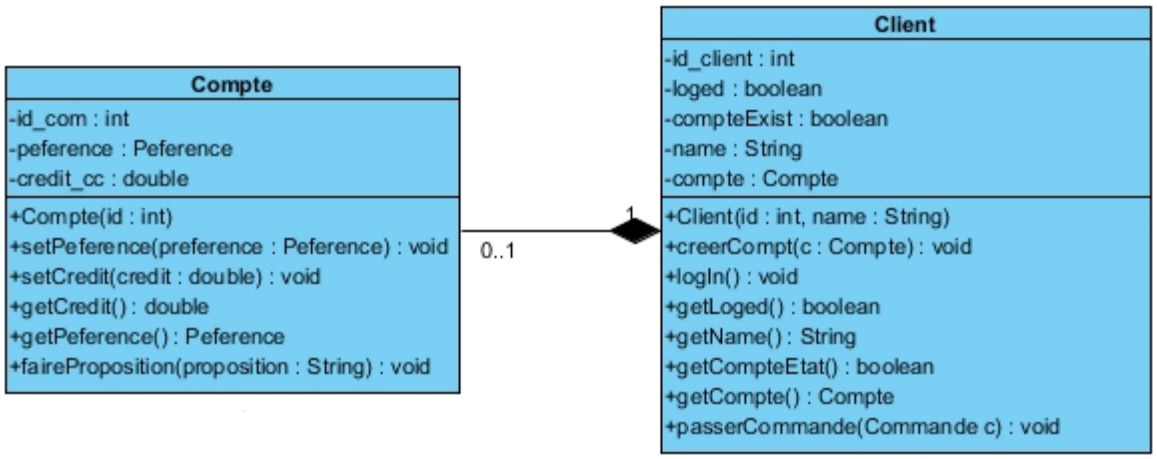


Figure 6 : Relation de *Client-compte*

- **Relation de *Commande-boutique,* de *Commande-recette,* de *Commande-infopayment***

Si un client a créé un commande, il faut obligatoire avoir un lieu de récupération, choisir une recette et donner ses informations de paiement. Et un boutique ou une recette peut dans plusieurs commandes différentes.

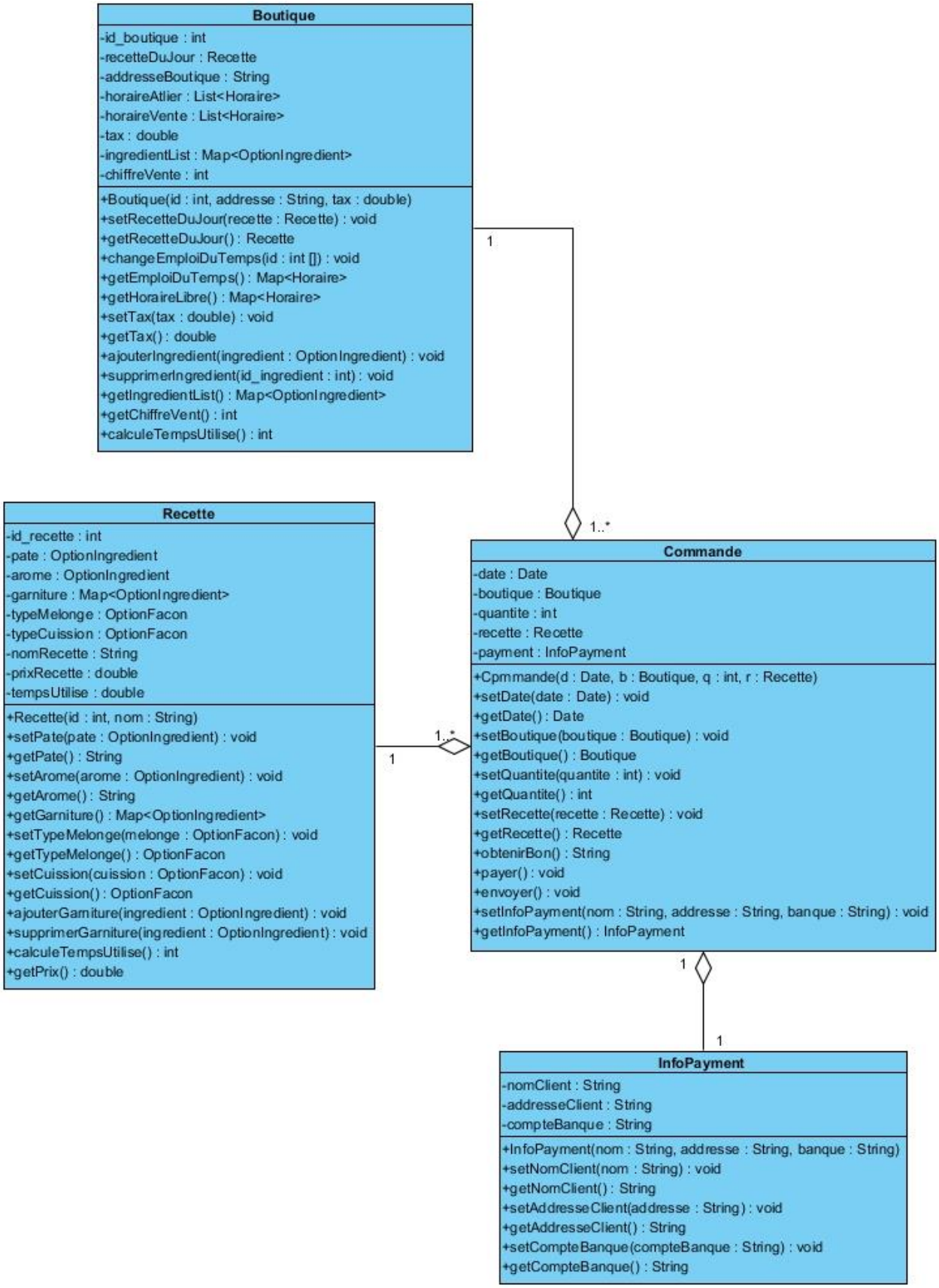


Figure 7 : Relation de *Commande-boutique*, de *Commande-recette*, de *Commande-infopayment*

- **Relation de *Compte-peference***

Si le client crée un compte, il pourrait enregistrer ses préférences dans son compte, mais ce n'est pas obligatoire.

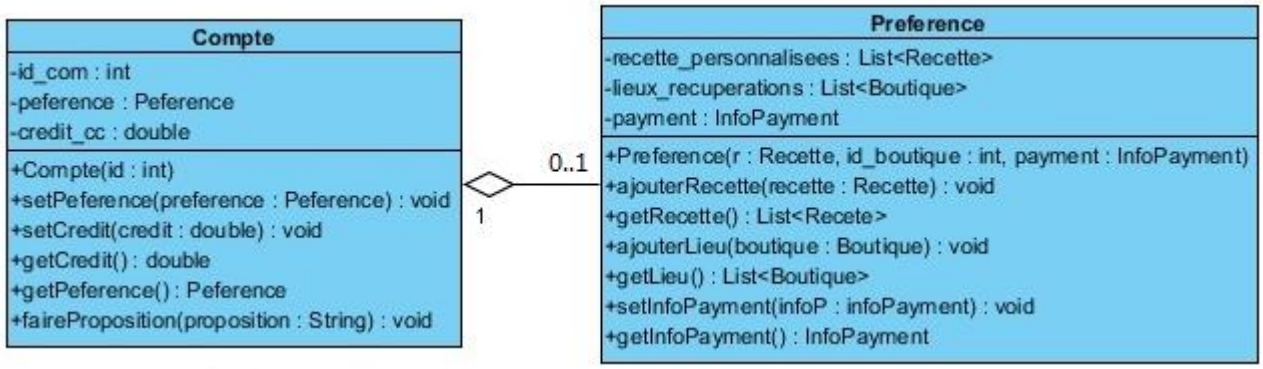


Figure 8 : Relation de *Compte-peference*

- **Relation de *Boutique-recette***

Les responsables de magasin peuvent définir la recette du "Today's special", alors chaque boutique doit avoir une recette du jour. Et une même recette peut être présentée dans plusieurs boutique.

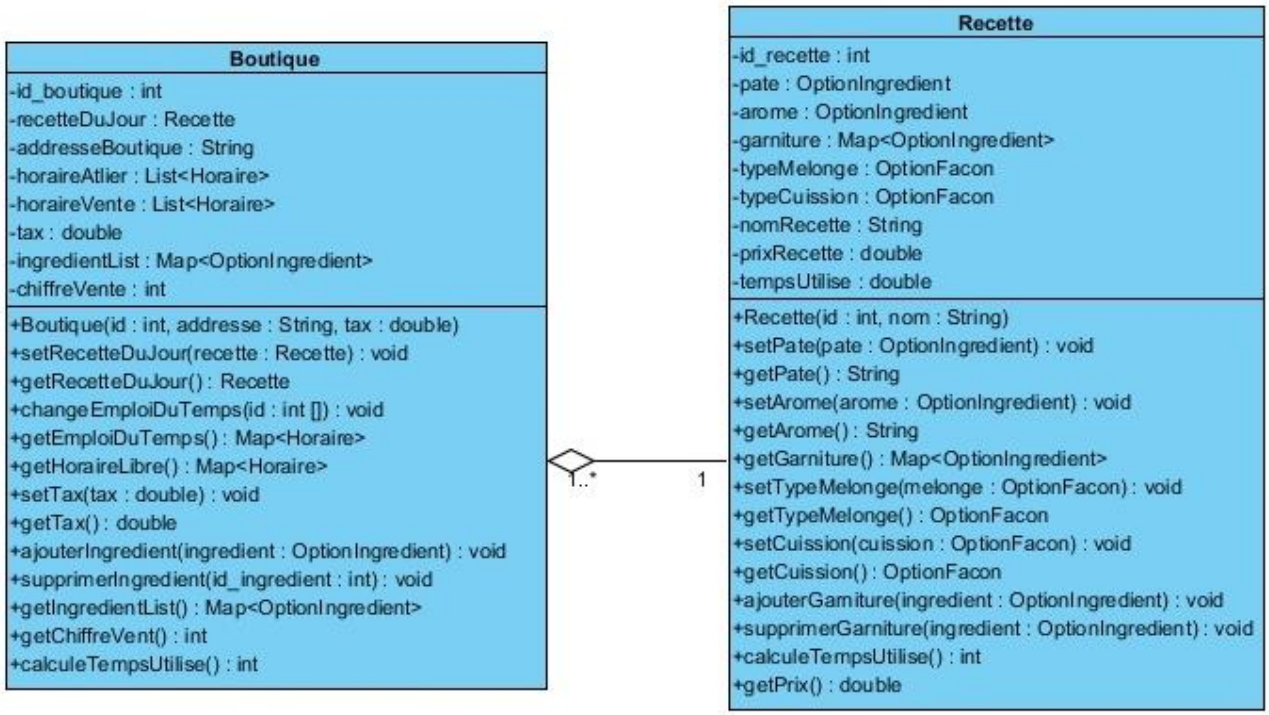


Figure 9 : Relation de *Boutique-recette*

- **Relation de *Boutique-horaire***

Grâce ànotre structure de la classe Horaire, chaque boutique a plusieurs horaires de vente et d'atelier.

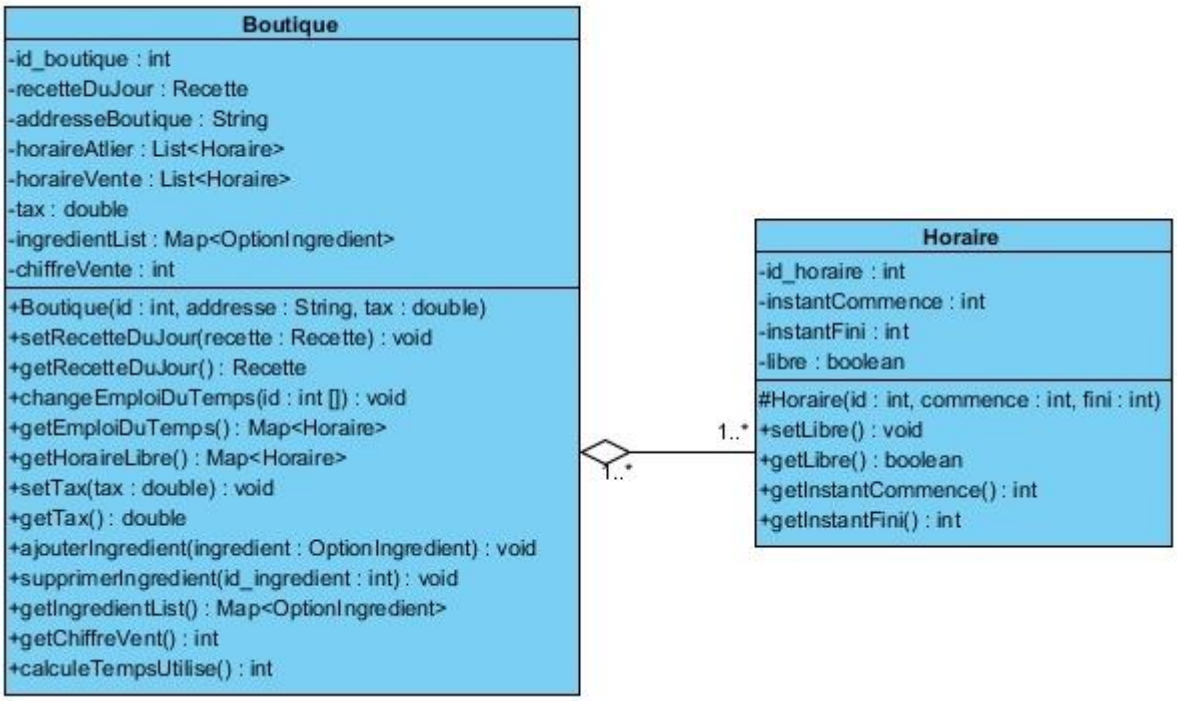


Figure 10 : Relation de *Boutique-horaire*

- **Relation de *Boutique-optionIngredient***

Chaque boutique a des ingrédients différents, alors chaque boutique a un list d'ingrédients et un ingrédient peut dans quelques boutiques différents.

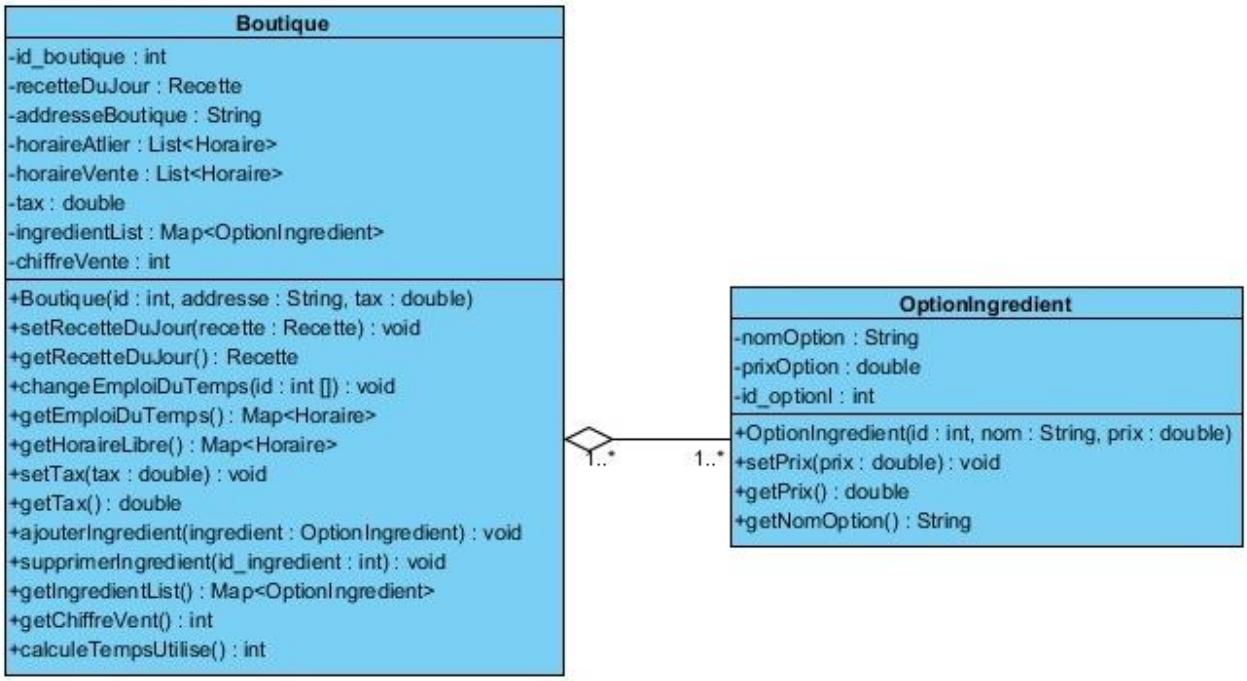


Figure 11 : Relation de *Boutique-optionIngredient*

- **Relation de *Societe-boutique***

Un société a plusieurs boutiques.

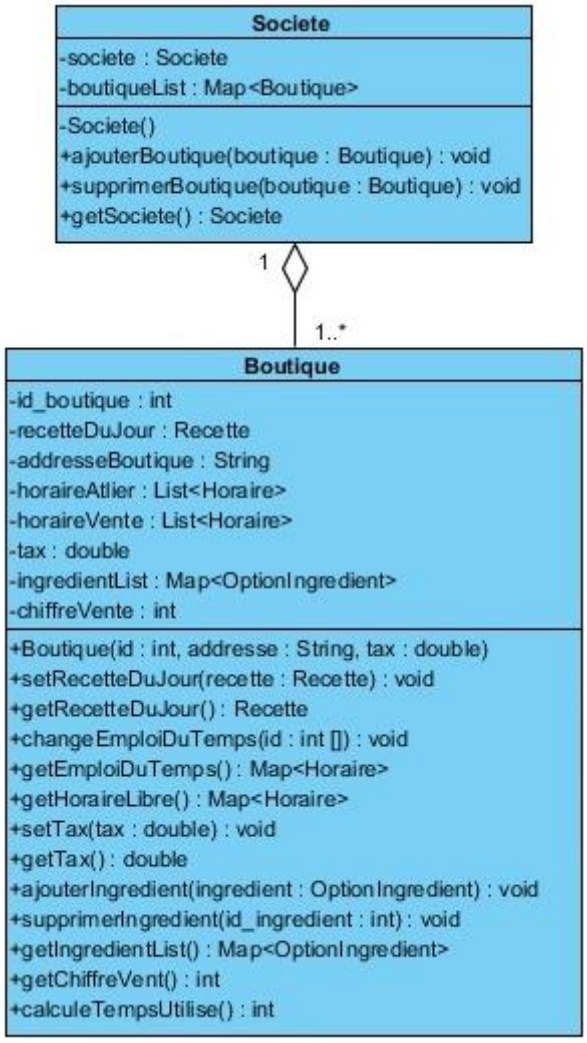


Figure 12 : Relation de *Société-boutique*

- **Relation de *Preference-boutique*, de *Preference-recette*, de *Preference-infopayment***
  Pour la préférence d'un utilisateur, il pourrait enregistrer plusieurs boutiques préférées, plusieurs recettes préférées, et une informaton de paiement. Et une recette ou une boutique pourrait être enregistrée dans plusieurs préférences.

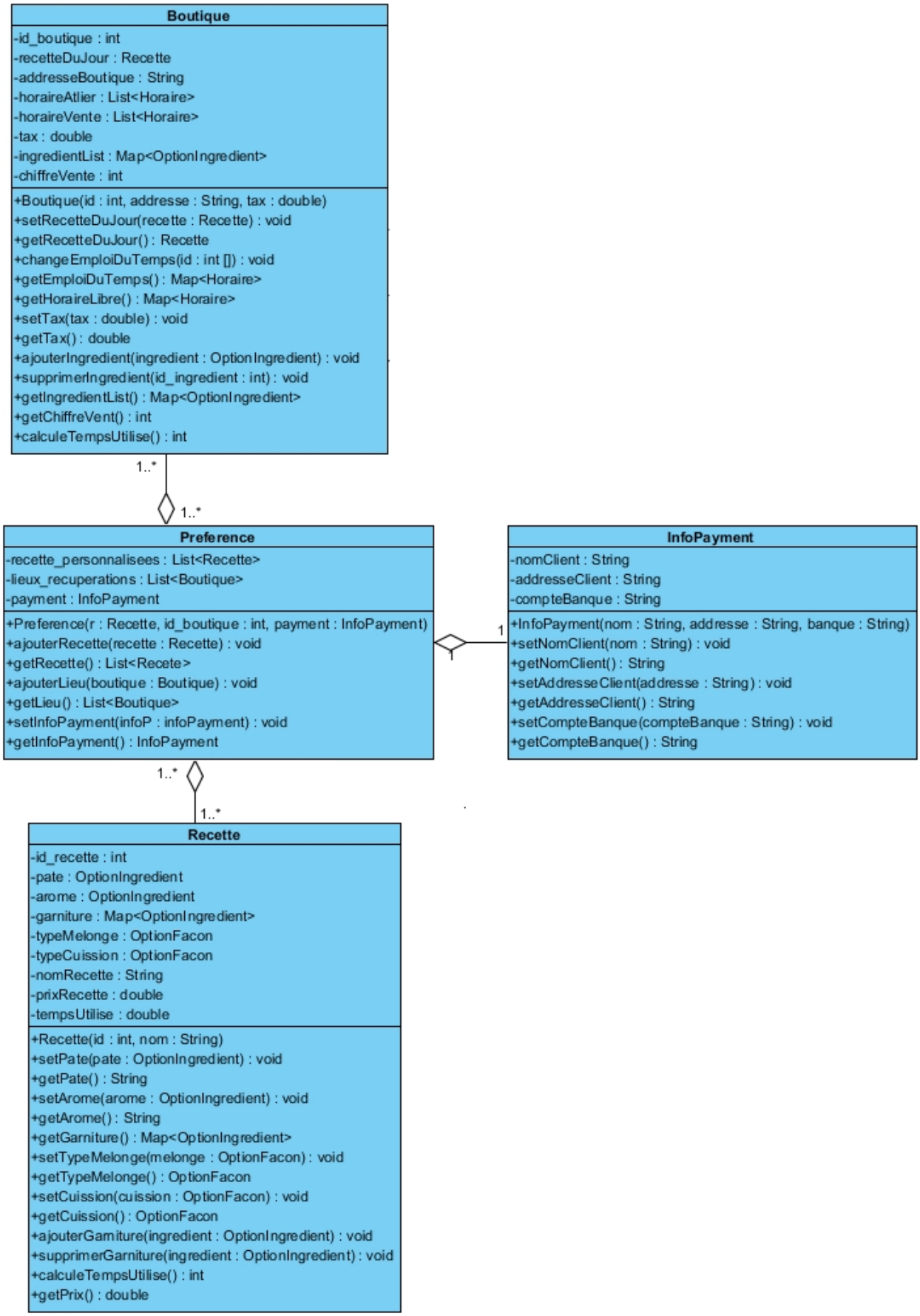


Figure 13: Relation de *Preference-boutique*, de *Preference-recette*, de *Preference-infopayment*

- **Relation de *Recette-optionIngredient*, de *Recette-optionfacon***

Dans une recette, elle contient plusieurs ingrédients différentes et plusieurs types de cuisson[2]. En revanche, les ingrédients et les types de cuisson pourraient aussi appartenir dans plusieurs recettes.

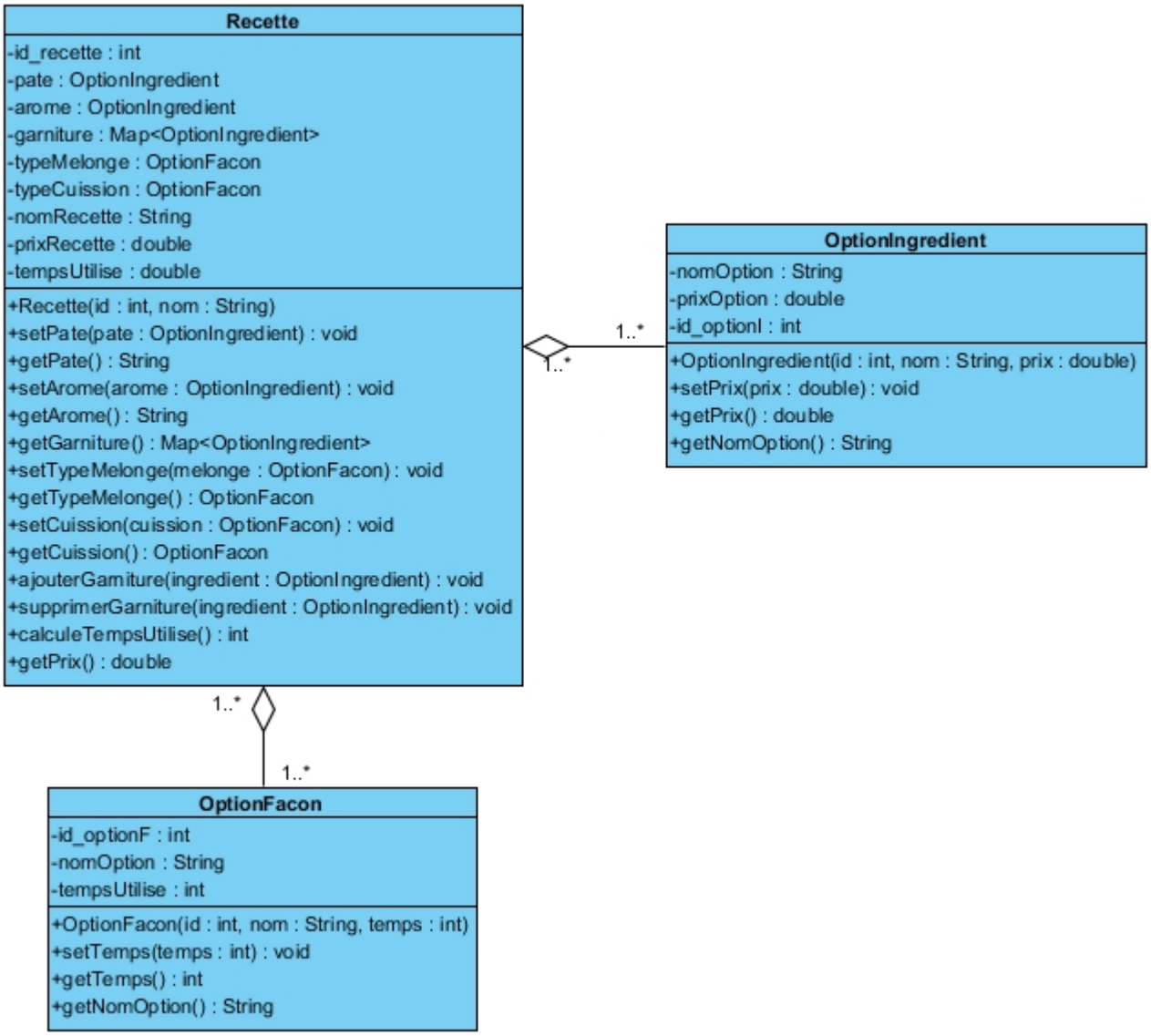


Figure 14: Relation de *Recette-optionIngredient*, de *Recette-optionfacon*

[2] Les cuissons contiennent le type de mélange et le type de cuisson.

Commande
-date : Date
-boutique : Boutique
-quantite : int
-recette : Recette
-payment : InfoPayment
+Cpmmande(d : Date, b : Boutique, q : int, r : Recette)
+setDate(date : Date) : void
+getDate() : Date
+setBoutique(boutique : Boutique) : void
+getBoutique() : Boutique
+setQuantite(quantite : int) : void
+getQuantite() : int
+setRecette(recette : Recette) : void
+getRecette() : Recette
+obtenirBon() : String
+payer() : void
+envoyer() : void
+setInfoPayment(nom : String, addresse : String, banque : String) : void
+getInfoPayment() : InfoPayment

CommandeFinder
+findById(id : int) : Commande
+findByDate(date : Date) : List<Commande>
+findByQuantite(quantite : int) : List<Commande>
+findByBoutique(id_boutique : int) : List<Commande>

CommandeGateway
+insert(c : Commande) : void
+update(c : Commande) : void
+delete(c : Commande) : void
+getRecette(id : int) : Recette

Client
-id_client : int
-loged : boolean
-compteExist : boolean
-name : String
-compte : Compte
+Client(id : int, name : String)
+creerCompt(c : Compte) : void
+logIn() : void
+getLoged() : boolean
+getName() : String
+getCompteEtat() : boolean
+getCompte() : Compte
+passerCommande(Commande c) : void

InfoPayment
-nomClient : String
-addresseClient : String
-compteBanque : String
+InfoPayment(nom : String, addresse : String, banque : String)
+setNomClient(nom : String) : void
+getNomClient() : String
+setAddresseClient(addresse : String) : void
+getAddresseClient() : String
+setCompteBanque(compteBanque : String) : void
+getCompteBanque() : String

Compte
-id_com : int
-peference : Peference
-credit_cc : double
+Compte(id : int)
+setPeference(preference : Peference) : void
+setCredit(credit : double) : void
+getCredit() : double
+getPeference() : Peference
+faireProposition(proposition : String) : void

CompteFinder
+finById(id : int) : Compte
+findByClient(id_client : int) : Compte

CompteGateway
+insert(c : Compte) : void
+update(c : Compte) : void
+delete(c : Compte) : void
+getClient(id_compte : int) : Client
+getCredit(id_compte : int) : double

Preference
-recette_personnalisees : List<Recette>
-lieux_recuperations : List<Boutique>
-payment : InfoPayment
+Preference(r : Recette, id_boutique : int, payment : InfoPayment)
+ajouterRecette(recette : Recette) : void
+getRecette() : List<Recete>
+ajouterLieu(boutique : Boutique) : void
+getLieu() : List<Boutique>
+setInfoPayment(infoP : infoPayment) : void
+getInfoPayment() : InfoPayment

RecetteFinder
+findById(id : int) : Recette
+findByName(name : String) : List<Recette>
+findByPrix(prix : double) : List<Recette>
+findByTempsUtilise(temps : int) : List<Recette>

Recette
-id_recette : int
-pate : OptionIngredient
-arome : OptionIngredient
-garniture : Map<OptionIngredient>
-typeMelonge : OptionFacon
-typeCuission : OptionFacon
-nomRecette : String
-prixRecette : double
-tempsUtilise : double
+Recette(id : int, nom : String)
+setPate(pate : OptionIngredient) : void
+getPate() : String
+setArome(arome : OptionIngredient) : void
+getArome() : String
+getGarniture() : Map<OptionIngredient>
+setTypeMelonge(melonge : OptionFacon) : void
+getTypeMelonge() : OptionFacon
+setCuission(cuission : OptionFacon) : void
+getCuission() : OptionFacon
+ajouterGamiture(ingredient : OptionIngredient) : void
+supprimerGarniture(ingredient : OptionIngredient) : void
+calculeTempsUtilise() : int
+getPrix() : double

RecetteGateway
+insert(r : Recette) : void
+update(r : Recette) : void
+delete(r : Recette) : void
+getPate(id : int) : String
+getArome(id : int) : String
+getGarniture(id : int) : Map<OptionIngredient>
+getTypeMelonge(id : int) : OptionFacon
+getCuission(id : int) : OptionFacon
+getTempsUtilise(id : int) : int
+getPrix(id : int) : double

OptionFaconFinder
+findById(id : int) : OptionFacon
+findByTempsUtilise(tu : int) : List<OptionFacon>
+findByName(name : String) : List<OptionFacon>

OptionFacon
-id_optionF : int
-nomOption : String
-tempsUtilise : int
+OptionFacon(id : int, nom : String, temps : int)
+setTemps(temps : int) : void
+getTemps() : int
+getNomOption() : String

OptionFaconGateway
+insert(of : OptionFacon) : void
+update(of : OptionFacon) : void
+delete(of : OptionFacon) : void

Console
+run()

ConsoleClient
+run()

ConsoleSociete
+run()

Societe
-societe : Societe
-boutiqueList : Map<Boutique>
-Societe()
+ajouterBoutique(boutique : Boutique) : void
+supprimerBoutique(boutique : Boutique) : void
+getSociete() : Societe

OptionIngredient
-nomOption : String
-prixOption : double
-id_optionI : int
+OptionIngredient(id : int, nom : String, prix : double)
+setPrix(prix : double) : void
+getPrix() : double
+getNomOption() : String

OptionIngredientGateway
+insert(oi : OptionIngredient) : void
+update(oi : OptionIngredient) : void
+delete(oi : OptionIngredient) : void

OptionIngredientFinder
+findById(id : int) : OptionIngredient
+findByPrix(prix : double) : List<OptionIngredient>
+findByName(name : String) : List<OptionIngredient>

HoraireGateway
+insert(h : Horaire) : void
+update(h : Horaire) : void
+delete(h : Horaire) : void

HoraireVente
+HoraireVente(id : int, commence : int, fini : int)

Horaire
-id_horaire : int
-instantCommence : int
-instantFini : int
-libre : boolean
#Horaire(id : int, commence : int, fini : int)
+setLibre() : void
+getLibre() : boolean
+getInstantCommence() : int
+getInstantFini() : int

HoraireAtlier
-capacite : int
-reste : int
+HoraireAtlier(id : int, commence : int, fini : int)
+HoraireAtlier(id : int, commence : int, fini : int, capacite : int)
+getCapacite() : int
+setCapacite(c : int) : void
+getReste() : int

HoraireFinder
+findById(id : int) : Horaire
+findByInterval(Intervale : String) : HashSet<int, Horaire>
+findByBoutiqueLibre(id_boutique) : List<Horaire>

BoutiqueFinder
+findById(id : int) : Boutique
+findByAddresse(add : String) : Boutique
+findByChiffreVente(cv : int) : List<Boutique>

Boutique
-id_boutique : int
-recetteDuJour : Recette
-addresseBoutique : String
-horaireAtlier : List<Horaire>
-horaireVente : List<Horaire>
-tax : double
-ingredientList : Map<OptionIngredient>
-chiffreVente : int
+Boutique(id : int, addresse : String, tax : double)
+setRecetteDuJour(recette : Recette) : void
+getRecetteDuJour() : Recette
+changeEmploiDuTemps(id : int []) : void
+getEmploiDuTemps() : Map<Horaire>
+getHoraireLibre() : Map<Horaire>
+setTax(tax : double) : void
+getTax() : double
+ajouterIngredient(ingredient : OptionIngredient) : void
+supprimerIngredient(id_ingredient : int) : void
+getIngredientList() : Map<OptionIngredient>
+getChiffreVent() : int
+calculeTempsUtilise() : int

BoutiqueGateway
+insert(b : Boutique) : void
+update(b : Boutique) : void
+delete(b : Boutique) : void

3.2. Object-Relational Mapping

Grâce aux relations entre les classes au-dessus, nous avons modéliser la base de données.

Nous avons décidé avant toute chose de mettre un attibut id à toutes nos classes qui doivent être stockées dans la base de données pour pouvoir représenter chaque instance de manière unique.

Nous avons premièrement choisi les données qui doivent être stockées dans la base de données, après puisque la classe nous présente une relation de table dans la base de données, nous pouvons modéliser les tables grâce à chaque classe correspondante.

Nous avons commencé par la classe *Client*. Nous savons que n'importe qui peut faire la commande, alors, les informations d'utilisateur peuvent être stockées dans la commande. Et nous savons aussi qu'un compte doit appartenir au utilisateur, mais pas tous les utilisateur ont un compte. Alors, il ne faut pas modéliser la table de client. Ensuite, parce que la relation entre la classe *Client* et la classe *Compte* est 1-1, alors nous avons utilisé la solution de ***relation merge*** pour mettre le compte et son prossesseur(id_client, code) ensemble.

Compte		
id_compte	**Integer(10)**	**Unique**
credit_carte_cadeau	Double	
id_client	Integer(10)	
code	varchar(16)	

(les clés sont représentées en gras)

Dans un compte, il peut aussi avoir une préférence, puisque la relation entre le *Compte* et la *Préférence* est 1-0..1. Donc, nous avons ajouté un ***Foreign Key*** (id_compte)dans la table de préférence. Ensuite, la Préférence et la Commande contiennent une information de paiement, les relations entre eux sont 1-1. Donc, nous avons ajouté un autre ***Foreign key***(id_info_payment) dans la table de *Préférence* et aussi dans la table de *Commande*.

Preference		
id_preference	**Integer(10)**	**Unique**
id_compte		ref Compte
id_info_payment		ref InfoPayment

InfoPayment		
id_info_payment	**Integer(10)**	**Unique**
nom_client	varchar(30)	
addresse_client	varchar(50)	
compte_banque	varchar(30)	

Parce que le système doit compter le chiffre de vente, et nous n'avons pas la table d'utilisateur, alors une table de *Commande* est nécessaire. Avec la relation entre la classe *Commande* et la classe *Boutique*, la relation entre la classe *Commande* et la classe *Recette*, nous avons ajouté deux ***Foreign***

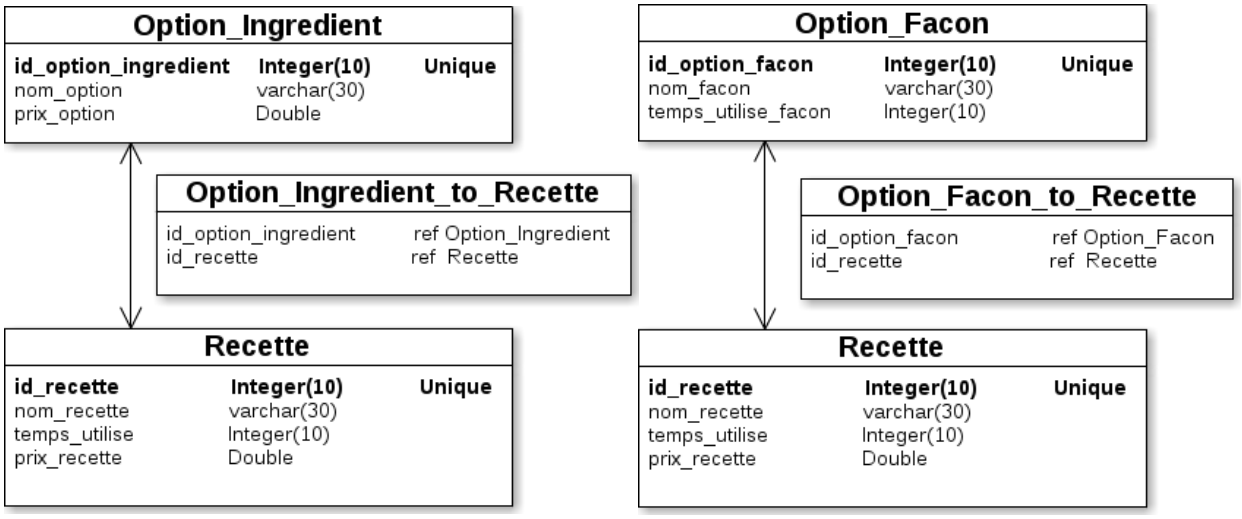
Keys(lieu_recuperation et recette_demande) dans la table de *Commande*.

Commande		
id_commande	**Integer(10)**	**Unique**
date	Date	
lieu_recuperation		ref Boutique
quantite	Integer(10)	
recette_demande		ref Recette
id_infopayment		ref InfoPayment

Considérons de notre diagramme de classe, une table de *Recette* est aussi nécessaires. Parce que, le système doit stocker les recettes que les utilisateur ont enregistré.

Recette		
id_recette	**Integer(10)**	**Unique**
nom_recette	varchar(30)	
temps_utilise	Integer(10)	
prix_recette	Double	

Mais, dans la classe Recette, il y a aussi les ingrédients et le type de cuisson. Pour trouver les informations des ingrédients et du type de cuisson facilement, nous avons aussi modéliser la table de ingrédient qui contient les pâtes, les arômes et les garnitures(*OptionIngredient)* et la table de cuisson qui contient le type de mélange et le type de cuisson(*OptionFacon)*. Parce que, une recette a plusieurs ingrédients et un ingrédient peut appartient dans plusieurs recette, alors leur relation est M-N. Donc, nous avons utilisé la solution de ***Association Table***(Option_Ingredient_to_Recette). Et en ce qui concerne la relation entre *Recette* et *OptionFacon*, c'est la même chose(Option_Facon_to_Recette).



Nous pensons qu'une table de *Boutique* est aussi nécessaire, parce que le système peut facilement trouver les informations de boutique, par exemple sa taxe ou sa recette du jour, et renouveler les informations de chaque boutique.

Boutique		
id_boutique	**Integer(10)**	**Unique**
recette_du_jour		ref Recette
addresse_boutique	varchar(50)	
tax	Double	
chiffre_vente	Integer(10)	

Ensuite, il faut bien expliquer la construction de notre table de *Horaire_Vente* et de *Horaire_Atlier*.

Horaire_Atlier		
id_horaire_atlier	**Integer(10)**	**Unique**
instant_commence	Integer(10)	
instant_fini	Integer(10)	
libre	Boolean	
capacite_maximale	Integer(10)	
capacite_reste	Integer(10)	
jour	Integer(10)	
id_boutique		ref Boutique

Horaire_Vente		
id_horaire_vente	**Integer(10)**	**Unique**
instant_commence	Integer(10)	
instant_fini	Integer(10)	
libre	Boolean	
jour	Interger(10)	
id_boutique		ref Boutique

Grâce à la table de *Recette*, nous croyons que le temps pour fabriquer les cookies ayant le type de cuisson différente n'est pas pareil. En plus, la quantité de commande est aussi un facteur d'influence pour le temps utilisé. Alors, pour le système, ce n'est pas facile de dresser l'horaire automatiquement. Donc, nous stockons les intervals de temps raisonnables dans la table. Le système peut ajouter et renouveler les informations facilement. La relation entre la table *Boutique* et la table *Horaire_Atlier* est 1-N, parail pour la table *Boutique* et la table *Horaire_Vente*. Nous utilisons la solution de ***Foreign Key***.

Par exemple, soit la cuisson la plus complexe utilise une heure. Nous stockons les données dans la table *Horaire_Atlier* comme au-dessous.

id_horaire_atlier	instant_commence	instant_fini[3]	capacité_maximale	capacité_reste	libre	jour[4]	id_boutique
1	0900	1000	300	0	false	1	1
2	1000	1100	300	20	true	1	2
3	1100	1200	300	300	true	2	2

[3] ***instant_commence***,***instant_fini***: 0900 est neuf heures, 1000 est dix heures, 1030 est dix heures et demi

[4] ***jour***: 1.lundi, 2.mardi, 3 mercredi, etc

Supposons un utilisateur a command é 20 cookies, et il veut le prendre à mardi dans la boutique 2. Alors, le système peut l'ajouter à la deuxème horaire(*id_horaire_atlier*=2) pour les frabriquer.

Donc, notre structure global de la base de donn ée est au-dessous.

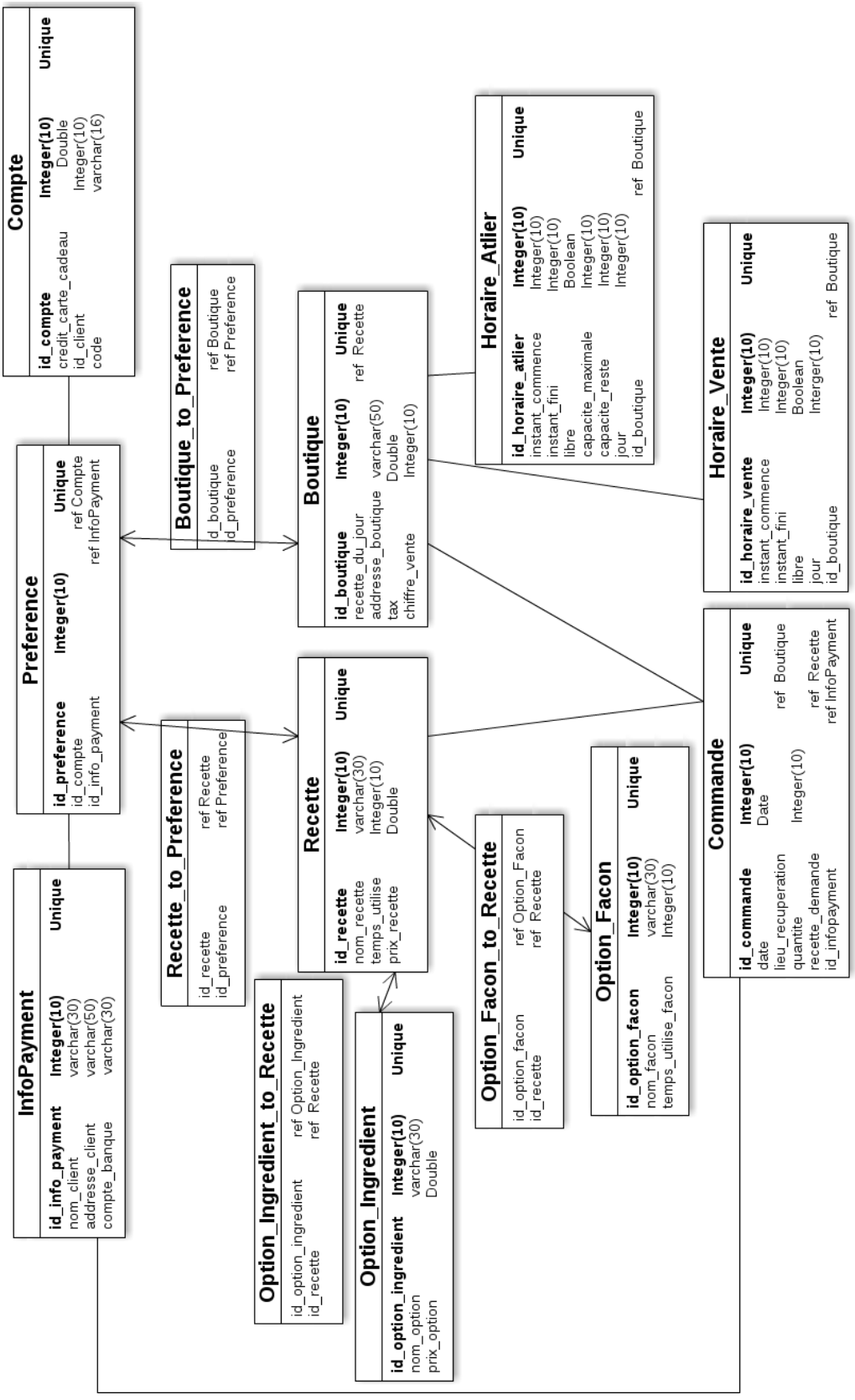


Figure 16: Structure global de la base de donn ée

3.3. Architecture des composants

Voil à le diagramme de composants:

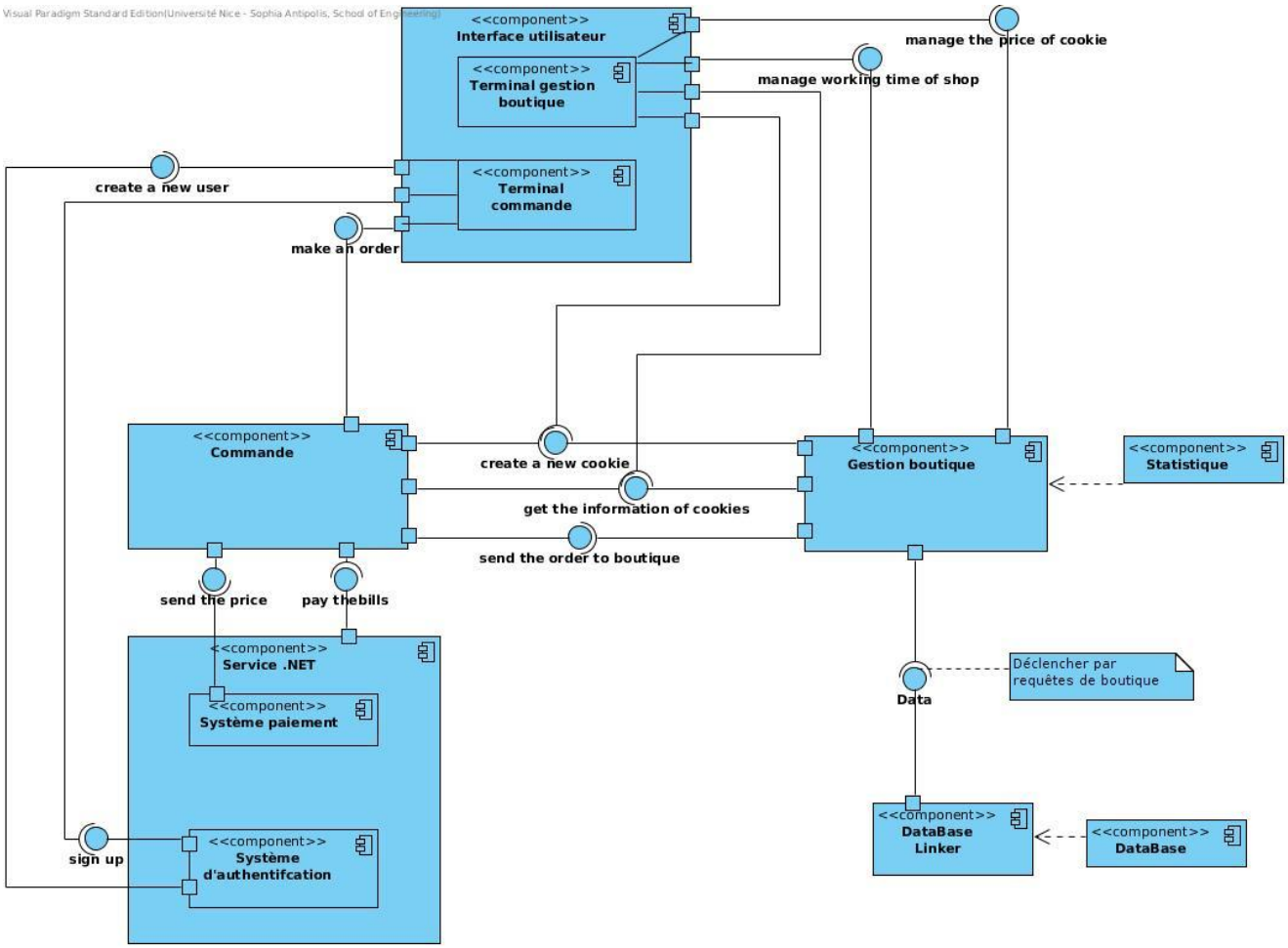


Figure 17 : Architecture des composants

Nous avons tout d'abord l'**Interface utilisateur** qui permet d'identifier la type d'utilisateur:
Soit un client utilisant le composant **terminal commande** qui peut *creer une nouvelle commande ou un nouveau compte ou s'authentifier* par le composant **Syst ème d'authentifcation**
Soit un responsable de boutique utilisant le composant **terminal gestion boutique** qui permet de *gérer l'horaire de boutique, créer une nouvelle recette sp éciale, afficher les informations des recettes* ou *changer les prix des ingr édiants*

Le composant **gestion boutique** recup ère des informations de base de donn ées et fournit des interfaces dessus utilis é par des clients et des responsbles des boutiques.
Il a besoin de donn ées et les met à disposition du **DataBase Linker** via des requ êtes par l'appel des m éthodes voulues dans les classes concern ées.

Le composant **DataBase Linker** représente la couche d'interopérabilité et permet donc d'accéder à la **DataBase**.

Le composant **statistique** peut récupérer les informations des commandes de boutique (par example la quantité, le prix changé, des recettes préférées etc.).

Le composant **commande** permet d'effectuer les différentes actions décrites dans le scénario d'utilisation de faire une commande(cf. faire des choix d'ingrédiant ou des recettes pré-existanctes, créer le bon commande etc.), puis l'envoit à la boutique choisie et envoit le prix au composant **Système paiement** afin de payer.

Les composants **Système d'authentifcation et Système paiement** sont fournit par le service Information Technology(IT) ustilisant exclusivement des technologies .Net.

3.4. Diagramme de déploiement

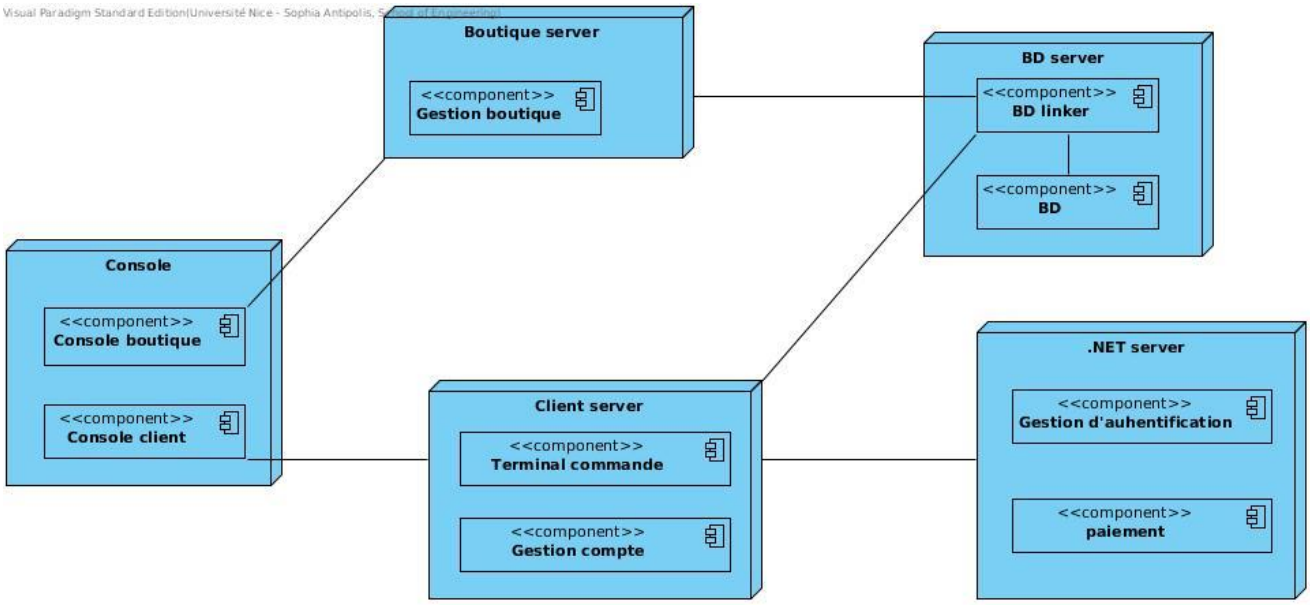


Figure 18 : Diagramme de déploiement

- Un serveur console héberge la plateforme d'accès: soit des clients, soit des responsables de boutique. On les sépare, car, si un des serveurs tombe, l'autre reste accessible et utilisable.
- Un serveur client. Cela permet l'authentification du client ou fait une commande.
- Un serveur boutique. Cela permet la gestion des information de boutique.
- La base de données est hébergée sur un serveur indépendant. le serveur client ou boutique peut accéder à la base de données pour récupérer les messages des channels externes. (Un système de sécurité fermant le flux entrant de requête pourra être mis en place.)