Shi Tianhao
Julien Le Van Suu

Rapport d’architecture

Introduction

L'application que nous allons créer fonctionne sur le principe du drive (commander ses courses sur le net puis aller les chercher déjà faites au supermarché), mais est spécifique à la vente de cookies.
De nombreux projets similaires ont vu le jour, ce qui donne une solide base d'expériences et de retours utilisateurs.
La plupart des fonctionnalités sont classiques, ce qui permet de créer une architecture assez simple en un nombre minimal de parties.

Le système sera décrit en quatre temps, allant du moins technique au plus technique(Modèle se rapprochant du 4+1, très adapté à ce genre d'application):
- Tout d'abord les scénarios qui nous permettront de déterminer des objectifs et des fonctionnalités minimales
- Ensuite la vue logique qui nous permettra de séparer les différentes fonctionnalités et leurs interactions.
- Puis, la vue de développement qui sera orientée du côté du développeur. Nous verrons dans cette partie des diagrammes de classe ainsi qu'un modèle relationnel évitant l'usage de SQL ou autre système de gestion de base de données.
- Pour finir, la vue physique qui nous permettra de montrer comment sera déployé notre système.

Sachant que des transactions financières auront lieu, il faudra assurer une sécurité maximale. Cependant, selon les spécifications, la partie qui doit être sécurisée est déjà séparée du reste, ce qui permettra de ne pas trop pénaliser la performance.
On peut constater que dans cette application les détails auront une importance capitale, on pourrait faire une application inutilisable en les maquant (ex : gestion des taxes, de la production, …).

Scénarios d'utilisation

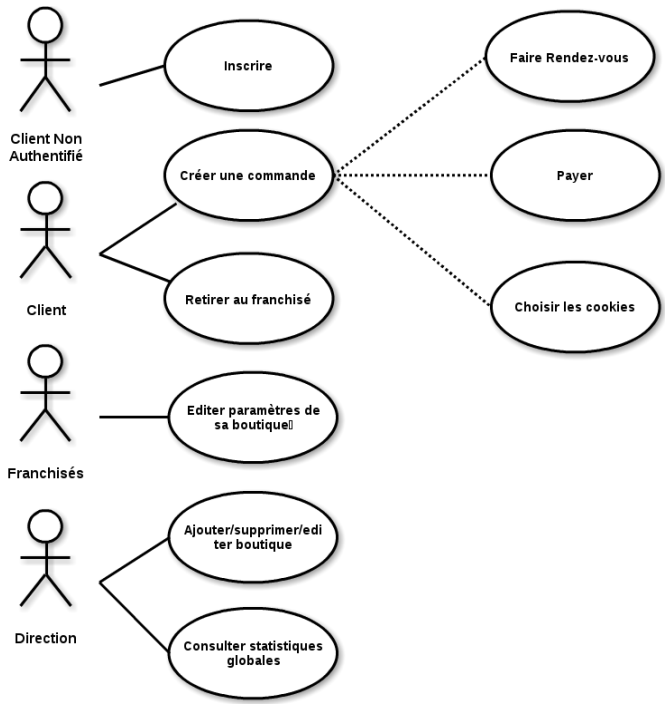
Avant de déterminer les différents scénarios, il peut être judicieux d'identifier les fonctionnalités principales.

Du coté utilisateur :

- Gestion de profil (gestion de préférences, programme de fidélité, carte cadeau)
- Divers choix quand aux recettes
- Gestion de commandes (commandes sans ou avec inscription préalable, …)
- Gestion de paiement

Du côté administrateur/franchisé :

- Gestion de boutique
- Gestion de prix (Il sera porté une attention particulière aux taxes dans le développement)
- Gestion des recettes et des ingrédients (on ne traitera pas les stocks ici)
- Capacité de production



Ces fonctionnalités nous amènent à définir des cas d'utilisation principaux.

Tous les scénarios impliquent qu'il y ait une liste en console du type

1. Choix 1
2. Choix 2
3. …
4. Revenir au menu précédent
5. Quitter

Votre choix ? |||

Les étapes marquées d'un * sont facultatives, elles sont réalisées au cas ou l'utilisateur entre son pseudo+mot de passe. Évidemment les scénarios ne sont pas exhaustifs, on se concentrera ici sur un consommateur/gestionnaire utilisant les fonctions basiques.

Utilisateur souhaitant commander une recette personnalisée.

- L'utilisateur se loggue *
- Le terminal lui affiche un message personnalisé et plusieurs options, notamment consulter son programme de fidélité, carte cadeaux, proposer de nouveaux ingrédients, ses préférences …*
- Il choisit l'option "commander" *
- Il choisit parmi les boutiques disponibles. Une fois choisie, il sera affiché les horaires de celles ci. (A noter que si l'utilisateur est loggué on remplit les champs avec ses préférences.*)
- On affiche ensuite deux choix, soit une recette spéciale, soit "à la carte"
- L'utilisateur choisit "à la carte"
- Il peut ensuite sélectionner plusieurs ingrédients (disponibles) en entrant leurs numéros séparés par un espace.
- Il choisit ensuite la date et l'heure(dans les plages autorisées,qui peuvent être restreintes selon l'affluence) à laquelle il viendra chercher sa commande.
- Le système accepte sa commande car elle n'excède pas les capacités de production.
- La commande est générée avec le prix mais non validée (éventuellement avec réduction *)
- Il peut la consulter (génération d'un document récapitulatif)
- Ensuite, il paye sur le même terminal (la partie donnée en .NET intervient en arrière plan)
- Une fois le paiement validé par un mécanisme qui reste à déterminer, la commande est validée (on régénère un document a l'attention du consommateur et du gestionnaire de boutique avec la mention "validé").

Utilisateur souhaitant choisir ses préférences.

- L'utilisateur se loggue
- Il choisit dans le menu "Préférences"
- Le système lui demande ensuite quelles est sa boutique préférée, puis ses recettes préférée, …
- Ses préférences sont enregistrées dans la base de données.

Utilisateur souhaitant s'inscrire.

- L'utilisateur s'inscrit avec un username et mot de passe
- Il accepte de voir collecter ses informations personnelles
- Les informations du client sont enregistrées dans le système d'authentification

Utilisateur donnant une carte cadeau.

- L'utilisateur se loggue
- Il choisit dans le menu "Offrir une carte cadeau"
- Le système lui demande le montant et l'utilisateur
- Une commande est générée puis validée (même procédure que pour le premier scénario)

Gestionnaire souhaitant changer un prix

- Le gestionnaire se loggue comme le ferait un client
- Il est redirigé vers une page spéciale
- Il choisit "Changer le prix d'un ingrédient"
- Il choisit le numéro de l'article a changer (une liste est affichée)
- Il change le prix
- Les changements sont enregistrés dans la base de données

Admin souhaitant ajouter une boutique

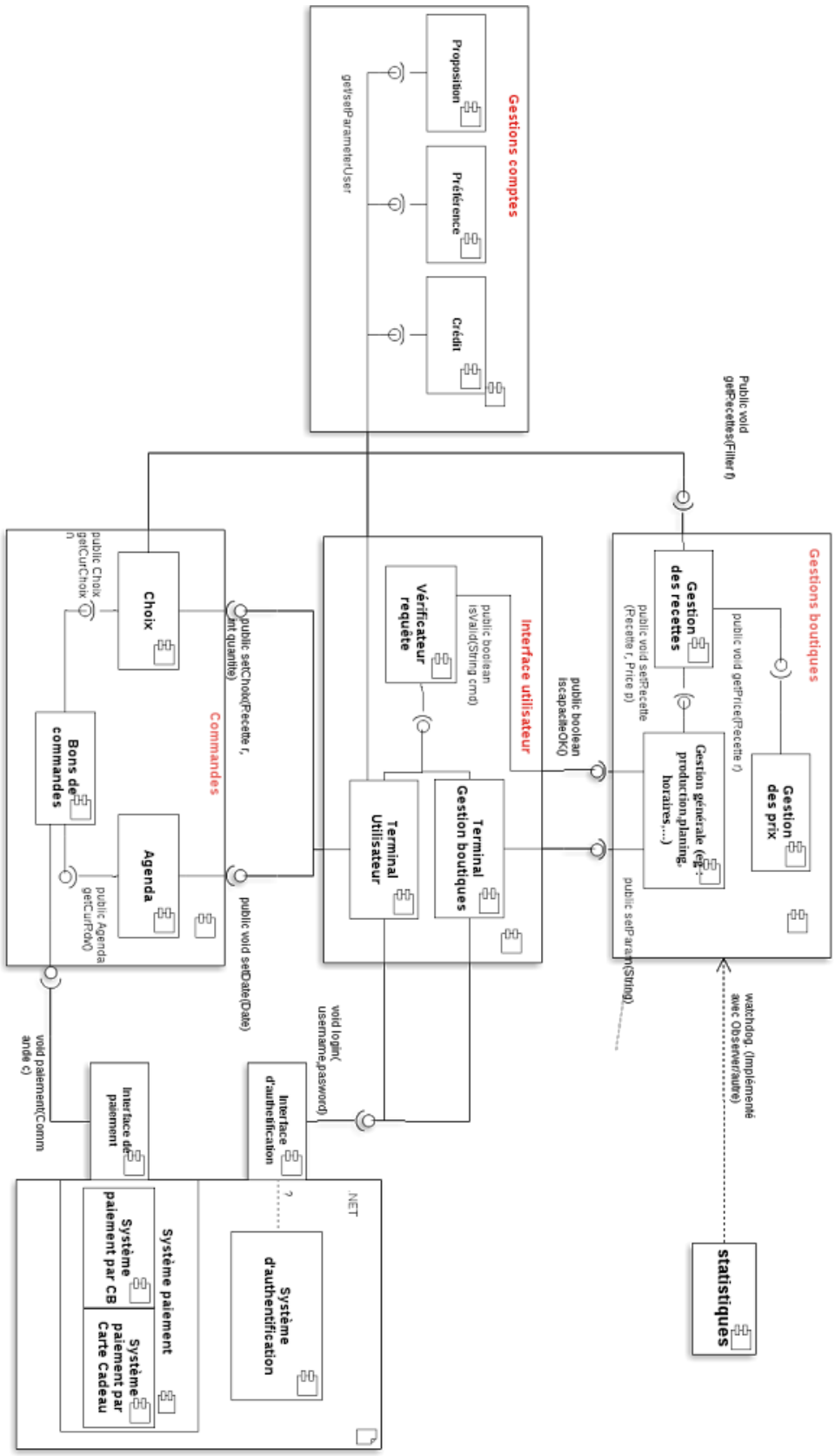
- L'admin se loggue avec le compte unique superuser
- Il choisit "Ajouter une boutique"
- Il entre les détails de celle ci(horaires, recettes disponibles, production, …)
- Il valide les données entrées
- La boutique est ajoutée a la base de données

On peut d'ores et déjà déterminer trois grandes parties : une impliquant le client, une autre impliquant le magasin et sa hiérarchie, et la dernière impliquant l'affichage (on essaiera de réutiliser cette partie pour les deux catégories d'utilisateurs).

Vue de développement

En accord avec la partie précédente, on peut imaginer les différentes parties et sous parties de notre application. L'objectif sera de créer des composants réutilisables, génériques et ayant une forte cohésion et un faible couplage avec les autres composants. L'accent sera mis sur la fonctionnalité et non la réalisation.

Certaines relations peuvent paraître étranges, ce pourquoi nous détaillerons chaque partie.


Gestions comptes
Proposition
Préférence
Crédit
getAndSetParameterUser
Public void getPrecettes(Filter f)
Gestions boutiques
Gestion des recettes
Gestion des prix
Gestion générale (cp, production,planing, horaires,...)
public void getPrice(Precette r)
public void setPrecette (Precette r, Price p)
public setParam(String)
Interface utilisateur
Vérificateur requête
public boolean isValid(String cmd)
Terminal Utilisateur
Terminal Gestion boutiques
public boolean iscapacitéeOK()
Commandes
Choix
public Choix getCurChoix ()
public setChoix(Precette r, int quantité)
Bons de commandes
Agenda
public Agenda getCurRdv()
public void setDate(Date)
void paiement(Comm ande c)
void login(username,password)
Interface de paiement
Interface d'authentification
watchdog (implémenté avec Observer/auteur)
statistiques
.NET
Système paiement
Système paiement par CB
Système paiement par Carte Cadeau
Système d'authentification

Tout d'abord, le module statistique fonctionnera à l'aide d'un Observer ce qui évite d'avoir des liens trop forts entre ce module et les autres. Il se contentera de surveiller les prix, les recettes choisies et la gestion du personnel. On peut déduire le nombre de commandes des informations précédentes (nombre de recettes = nombre de commandes).

Ensuite, il faudra séparer le terminal administrateur du terminal utilisateur. Cette différentiation sera faite au moment du login (Sous Linux, quand on est superutilisateur, le terminal affiche un #, ici ce sera le même principe). Les commandes disponibles à l'un et l'autre ne seront pas les mêmes, le seul facteur commun est la vérification de la validité d'une requête (bon numéro entré). Cela permet une plus grande sécurité mais ajoute du couplage. En effet, on constante qu'un lien est nécessaire entre le vérificateur de requêtes et la capacité maximale de production.
Si il n'y avait qu'un terminal pour les deux, ce problème ne se poserait pas.

De plus, nous avons identifié un lien entre le vérificateur de requête et la gestion générale : En effet, pour vérifier si une requête est valide on peut avoir besoin de la capacité de production. Cela ajoute du couplage mais permet que toutes les requêtes soient vérifiées de la même manière (vérificateur de requêtes générique).

Pour finir, un bon de commande est généré selon un rendez vous, un paiement, et un choix. Cela explique le nombre important de liens sur ce sous bloc, qui a besoin de toutes les informations possibles.

On utilisera pour se connecter au système .NET deux interfaces qui nous permettront de faire abstraction de cette partie dans la partie java. En revanche, il faudra veiller à maintenir un niveau de sécurité élevé dans ces interfaces car des informations critiques comme le numéro de carte ou d'user/login transiteront par ces interfaces.
L'interface se connectera comme demandé à l'aide de services webs. Le terme étant assez large, cette interface est tout de même nécessaire.

Le schéma ne comporte pas de composant base de données : et pour cause, ce n'est pas une fonctionnalité. Écrire et lire dans la base de données est inhérent à tous les composants dans le modèle NoSQL que nous allons mettre en place.

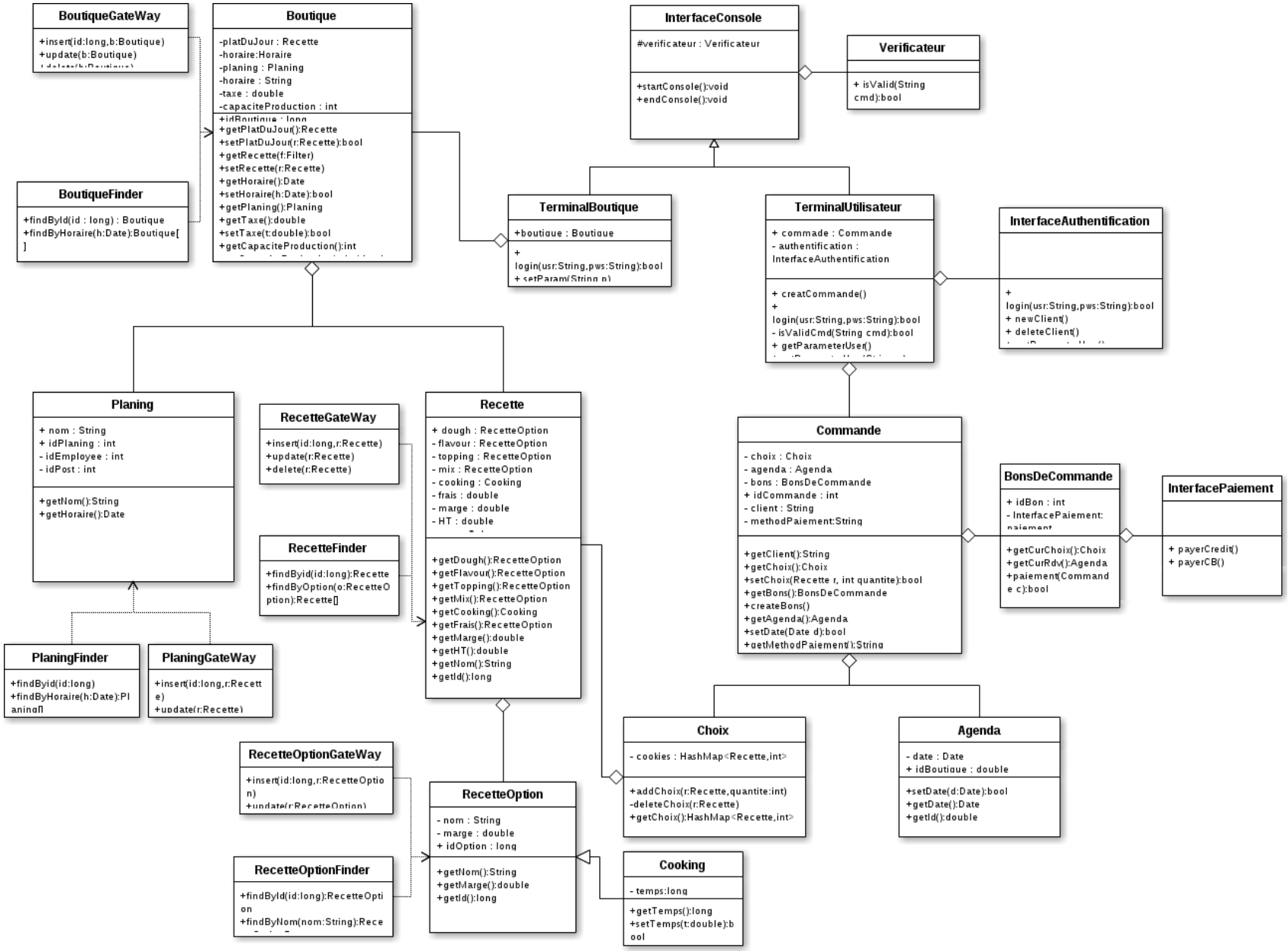
Pour conclure cette partie, on constate donc que les cinq modules qu'on a pu délimiter ont peu de dépendances entre eux, tout au plus trois. Cela permet de limiter le couplage et éventuellement de dégager des hiérarchies.

Vue logique

Nous allons dans cette vue détailler des aspects techniques qui permettront au développeur de mieux comprendre la vision globale du projet. Le but de cette partie est de montrer quelle sont les différentes relations entre les classes, quelles sont les relations objet bases de données,...

Modèle de données

Légende : Flèche: Héritage | Carre : Composition | Flèche pointillés : Observer



Shi Tinhao-Julien Le Van Suu | Sebastien Mosser, Serveurs d'entreprises | 03/2015

Nous avons choisi les *gateways* pour implémenter les manipulations de données : Cela nous paraît être l'architecture la plus adaptée pour ce projet(séparation de la partie métier et de la partie manipulation de données, au prix d'un plus fort couplage)
De plus, nous avons ajouté des *finders* dans le but d'éviter les méthodes statiques.
Notre choix en la matière est donc de découper au maximum le problème quitte à ajouter du couplage. Cependant, en faisant appel à la précédente vue, cela reste cohérent et justifiable : On aura un fort couplage à l'intérieur d'un même composant, mais un faible couplage si on se place à l'extérieur du composant.

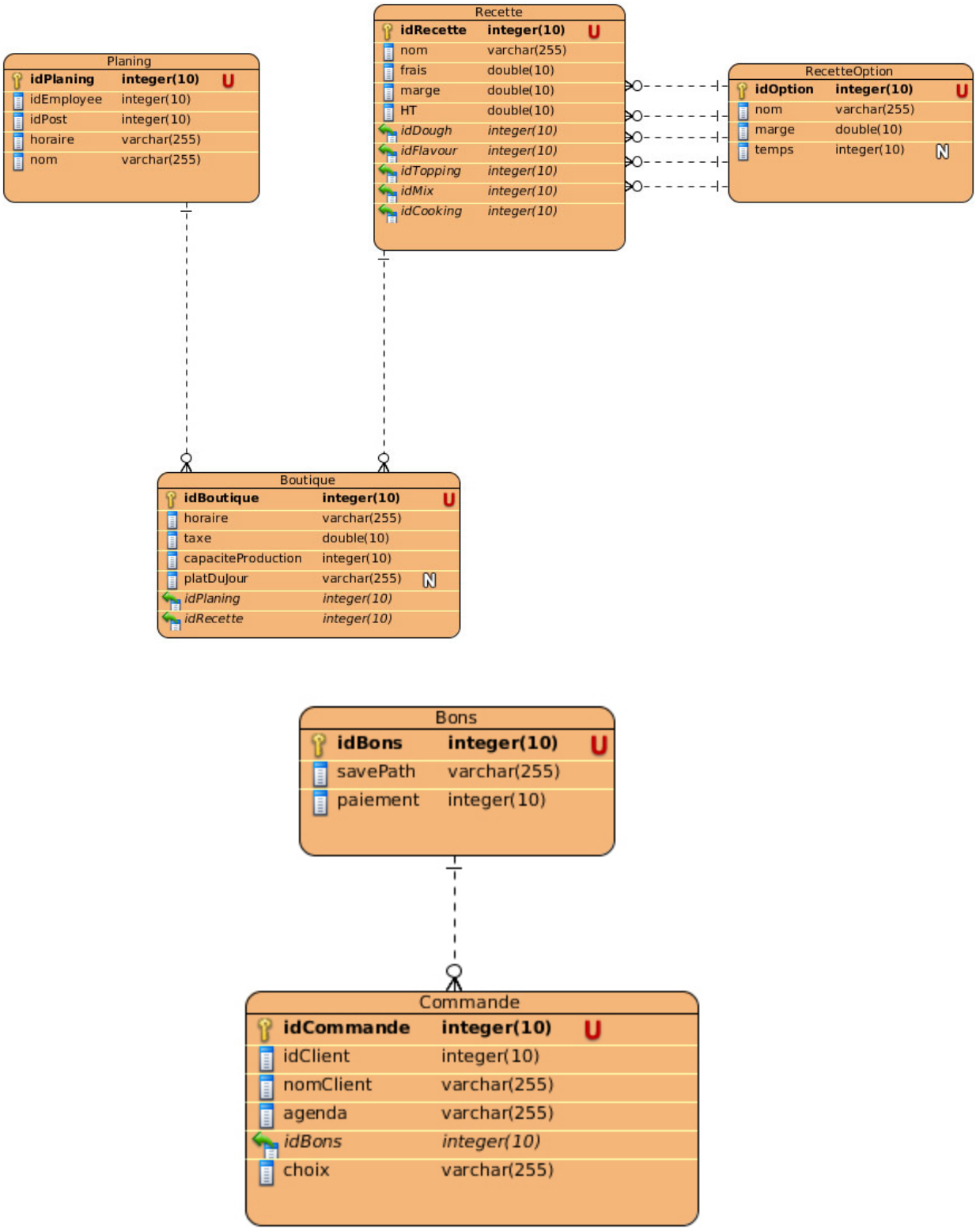
La *classe Choix* peut paraître inutile mais en fait elle se révélera utile pour faire des statistiques et garder en mémoire les différents choix des clients. De plus une HashMap permettra d'optimiser la vitesse de recherche.

Le fait d'ajouter une *recetteOption* peut être discutable. On remarque cependant que cela permet d'associer un nom a un identifiant (par exemple si c'est une saveur, l'identifiant commencera par "02"). Cela permet d'envisager, au prix d'une redondance, d'ajouter par la suite de nouvelles options (exemple : une option QtéSucre).
Cette classe fonctionne comme une plaque d'immatriculation, cela permet de retrouver facilement à partir d'un identifiant une chaîne de caractères.
Cooking est un cas particulier de recetteOption qui possède un certain temps.

Dans ce projet, il est difficile de dissocier les classes et leur équivalent relationnel : nous avons eu à l'idée ce couplage en créant notre diagramme de classes. Pour voir clair dans cette relation, nous allons présenter plusieurs tables correspondant à une classe.

Mapping objet - relationnel

Le schéma suivant montre quelles sont les relations entre les diverses tables.
Il ne prétend pas être exhaustif mais donne un aperçu de ces tables. Nous verrons ensuite plus en détail à quoi devront ressembler ces tables et leur contenu, et leur raison d'être.


Recette
idRecette integer(10) U
nom varchar(255)
frais double(10)
marge double(10)
HT double(10)
idDough integer(10)
idFlavour integer(10)
idTopping integer(10)
idMix integer(10)
idCooking integer(10)
Planing
idPlaning integer(10) U
idEmployee integer(10)
idPost integer(10)
horaire varchar(255)
nom varchar(255)
RecetteOption
idOption integer(10) U
nom varchar(255)
marge double(10)
temps integer(10) N
Boutique
idBoutique integer(10) U
horaire varchar(255)
taxe double(10)
capaciteProduction integer(10)
platDuJour varchar(255) N
idPlaning integer(10)
idRecette integer(10)
Bons
idBons integer(10) U
savePath varchar(255)
paiement integer(10)
Commande
idCommande integer(10) U
idClient integer(10)
nomClient varchar(255)
agenda varchar(255)
idBons integer(10)
choix varchar(255)

Nous avons choisi de modéliser le problème avec une seule table par hiérarchie d'héritage. Nous avons pu constater que dans le diagramme de classes les arbres d'héritages étaient très limités, donc cela n'aura guère d'influence sur l'espace pris par la table et cela n'engendra pas des tables d'une taille monstrueuse. En revanche, on garde les avantages de cette méthode, à savoir que le refactor n'influe pas sur nos tables, il n'y a pas besoin de faire des jointures et c'est une solution assez simple à penser et mettre en place.

De plus, nous n'avons pas fixé un modèle de relation unique pour tout le projet (exemple : Foreign Key partout, ou table intermédiaire partout). Cela serait contre productif car au sein d'un même projet on peut avoir besoin de ces relations, les trouver plus ou moins adaptées.

Nous allons présenter une table pour la classe *recette,* qui contiendra plusieurs champs intéressants.
On fera les recettes préfabriqués ou les préférences du client grâce à cet objet. Il ne paraît pas nécessaire de mettre un ID car il n'y aura pas énormément de recettes différentes.
En revanche dans le cas d'une recette personnalisée il faut absolument mentionner le pseudo du créateur de cette recette pour le retrouver ensuite.

Nom	Owner	dough	flavour	...	HT	marge
Soo Chocolate	Admin	Chocolate	Chili		2$	0.60$
Dark Tempation	Admin	OatMeal	Chili		3$	0.80$
MyCook	Raymond	Plain	Vanilla		1$	0.25$

Pour les *commandes*, il faudra veiller a donner un ID unique pour identifier la commande (le mieux serait quelque chose d'aléatoire afin que la commande ne soit pas falsifiable)

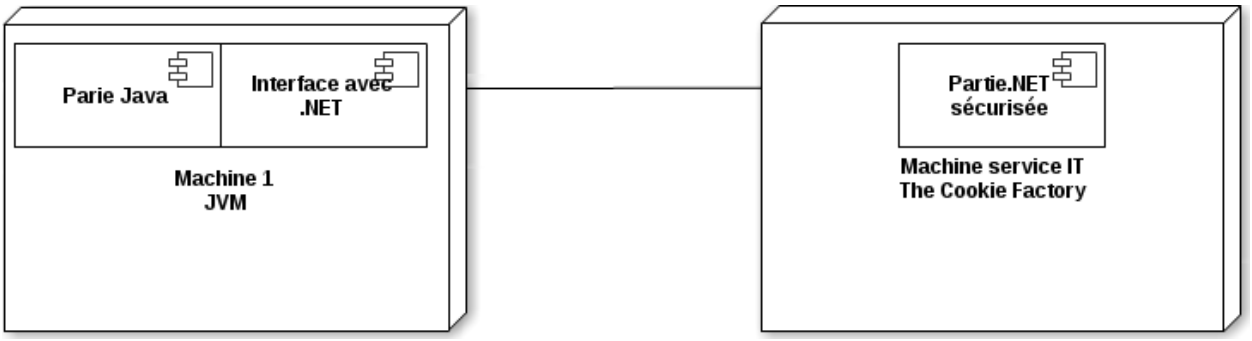
ID	Choix	Agenda	bons	client
4efgu7997	Choix1	[21/01 12H]	com1.txt	Tonton
4528852gt	Choix2	[21/01 15H]	com2.txt	Laurent
5scjcs7bkc	Choix3	[21/01 12H]	com3.txt	Michel

On constate donc une corrélation presque parfaite entre les classes et les tables; On est certes obligé de rajouter quelques champs pour faire le lien avec d'autres tables (Foreign Key dans le meilleur des cas, table intermédiaire dans le cas de la classe "Choix"), mais globalement on retrouve les mêmes structures.

Vue physique

Étant donné les spécifications du projet, il paraît plus convenable de distribuer le système sur deux machines. Une qui gérera la partie en Java, et une autre qui se chargera du .NET. En effet, le système fourni par le département IT de The Cookie Factory ne sera peut être pas dé localisable et ce serait même une faille pour la sécurité de l'entreprise. En séparant la partie "paiement/authentification" de la partie métier, on améliore grandement la sécurité. En revanche, il faut être conscient que cela peut poser des problèmes de performances (que se passe t-il si le réseau local est indisponible ?)

Le système physique est donc assez simple, nous avons deux machines qui communiquent entre elles.
La charge paraît répartie équitablement entre elles, car bien que la machine sécurisée doive traiter moins de requêtes, celles ci demanderont certainement un chiffrage très consommateur en puissance de calcul.



Conclusion

Nous avons choisi une architecture qui met le problème à plat : nous avons beaucoup de composition et peu d'héritage. Cela permet d'avoir des modules "atomiques", c'est à dire qui font une action très limitée, cela dans un but de réutiliser au maximum ceux ci. Le revers de la médaille et que cela introduit du couplage.
Dans l'optique d'avoir des relations avec des bases de données, ce modèle d'architecture est intéressant car il se marie avec les concepts des BDD sans aucun effort.
D'autres modèles, bien qu'ayant leurs avantages, peuvent nécessiter une gymnastique intellectuelle et technique et sont donc sources d'erreurs.