

# Projet Programmation Orientée Objet

COMPARATEUR INTERNE DE JEUX VIDEO  
ALEXANDRE GUITTON | JORDAN HOAREAU

I – Introduction.....	3
A – Présentation du projet .....	3
B - Les limites fixées pour ce projet.....	3
C – Choix stratégiques, pratiques et techniques.....	3
II – Les différentes phases du projet .....	4
A – Définition des critères .....	4
B – Définition des types de scores .....	4
C – Implémentation des scores .....	5
D – Implémentation de la demande et de l’offre.....	5
E – Définition de la base de données .....	5
1 – Choix du SGBD .....	5
2 – Modèle relationnel .....	5
F – Définition de l’IHM .....	6
III – Le projet d’un point de vue UML.....	7
A – Diagramme de cas d’utilisation.....	7
B – Diagramme de séquence.....	8
1 - Côté client .....	8
2 - Côté administrateur .....	9
C – Diagramme de classe perspective logicielle (noyau).....	9
D – Diagramme de package.....	12
E – Design Pattern .....	12
III – Le projet d’un point de vue fonctionnel.....	13
A – Les différents type de scores.....	13
1 – Score binaire.....	13
2 – Score Ecart.....	13
3 – Score Intervalle.....	14
4 – Score Keyword.....	14
5 – Score Multiple .....	15
6 – Score Multiple Complexe .....	15
7 – Score Style .....	16
B – Utilisation coté programmeur.....	17
1 – Comment ajouter un critère [Réutilisabilité] .....	17
2 - Création d’un type de score .....	17
3 - Ajout d’un élément à l’interface du Comparateur.....	18
C – Utilisation coté utilisateur .....	18
1 – Administration des offres .....	18

a – Connexion à l’interface d’administration .....	18
b – Suppression d’une offre .....	19
c – Création d’une nouvelle offre.....	19
d – Modification d’une offre .....	20
2 – Création d’une demande et réservation .....	21
V – Conclusion .....	23

## I – Introduction



---

### A – Présentation du projet

Dans le cadre du projet de Programmation Orientée Objet, en 4<sup>ème</sup> année à Polytech' Marseille, nous avons réalisé un Comparateur Interne sur le thème des jeux vidéo. L'objectif de ce projet était de mettre en application les différents principes, méthodes et techniques d'analyse, de modélisation, de conception et de programmation objet.

Nous avons choisi de prendre comme thème les jeux vidéo, puisqu'un jeu offre une grande gamme de caractéristiques servant de critère à notre application (en plus d'être un thème qui nous intéresse).

Le noyau à développer pour cette application devait permettre à partir d'une demande d'un utilisateur, de comparer l'ensemble des critères présents dans la demande à ceux de toutes les offres, pour en sortir une liste triée en fonction de la pertinence.

Nous avons ajouté à ce noyau développé en  **Java**<sup>™</sup>, une base de données  **SQLite** permettant de stocker les offres, mais aussi un ensemble d'éléments pour les critères. Une interface graphique [**SWING**] donnant à l'utilisateur une plus grande simplicité d'utilisation pour créer la demande, mais aussi à l'administrateur de gérer l'ensemble des offres.

Un fichier **{JSON}**, sert de fichier de configuration afin d'instancier certains critères et l'interface.

### B - Les limites fixées pour ce projet

Afin d'avoir assez de temps pour concevoir le noyau de l'application, nous avons pris 6 jeux arbitrairement, suffisamment différents les uns des autres afin de pouvoir comparer efficacement ceux-ci. Chaque jeu représente donc une catégorie d'offre précise : il ne s'agit pas ici de trier parmi une banque de données importante mais plutôt de créer une application sachant trier ces catégories de jeux en fonction d'une demande. Nous avons aussi choisi 14 critères regroupés en 7 catégories de critère. Chaque catégorie de critère possède sa méthode de calcul de score. Ainsi, nous avons pu nous centrer sur la création des algorithmes de calcul de score et finir en un temps assez court le noyau de l'application.

### C – Choix stratégiques, pratiques et techniques

Cette application étant portée sur un ensemble réduit d'offre et de critère, nous sommes partis du principe que cette application serait le squelette d'une application complète. Dans cette optique nous avons implémenté l'application en gardant comme idée principale : **la réutilisabilité** (une partie dans ce rapport est consacrée à l'ajout d'un nouveau critère). Celle-ci se retrouve dans l'utilisation d'un fichier de configuration (**config.json**) qui permet l'instanciation d'un grand nombre de paramètres, une base de données SQLite (qui est une base de données plus légère), ou encore la réflexivité utilisée dans l'IHM.

Dans cette optique nous avons centré l'application et plus particulièrement les critères autour des différents scores possibles. Ce qui permet d'avoir « éclaté » l'offre et la demande en ensemble de classe, représentant les critères, liées à des scores, et ainsi avoir un code très peu centralisé (grand nombre de classe) plus souple d'utilisation.

Dans un souci de cohérence nous verrons plus tard dans ce rapport que l'on a dû inverser la dépendance entre l'offre et la demande, puisqu'à l'origine l'offre dépendait de la demande, ce qui ne nous semblait pas naturel. Une offre peut exister sans une demande, mais une demande n'a pas de sens sans offre.

## II – Les différentes phases du projet

---

### A – Définition des critères

Lors de cette première phase nous avons fait un brainstorming afin d’avoir un nombre conséquent de critères. À l’issu de celui-ci, les critères trouvés étaient les suivants et répondent chacun à une problématique lors de l’achat d’un jeu vidéo :

- ❖ **Titre de jeu**
- ❖ **Editeur**
- ❖ **Description**
- ❖ **Note du jeu** : les utilisateurs ont-ils aimé le jeu ?
- ❖ **Date de sortie**
- ❖ **Genre – Catégorie de jeu** : s’agit-il d’un jeu de rôle, d’un jeu de combat, d’un jeu de course... ?
- ❖ **Prix**
- ❖ **Mode de jeu** : le jeu se joue-t-il en ligne ? Hors ligne ?
- ❖ **Difficulté**
- ❖ **Durée de vie**
- ❖ **Accessoires** : le jeu utilise-t-il des accessoires ?
- ❖ **Supports de jeu** : sur quelles consoles puis-je jouer à ce jeu ?
- ❖ **Type d’histoire** : univers à l’intérieur du jeu
- ❖ **Paielement** : Y-a-t-il un abonnement à payer régulièrement, une licence à acheter une fois ?

### B – Définition des types de scores

Une fois que nous avons nos critères, il a fallu les regrouper en catégories afin d’optimiser le temps passé à implémenter l’application ; avoir 14 scores à calculer étant assez chronophage, nous avons choisi de les regrouper en 7 types de scores différents, à savoir :

- ❖ **Keyword** : **titre** | **éditeur** | **description**  
Comparaison de mots-clés comme un moteur de recherche
- ❖ **Écart** : **durée de vie** | **difficulté**  
L’écart à la valeur choisie par l’utilisateur sera noté. (e.g. l’utilisateur choisi un jeu facile ; un jeu moyen aura un meilleur score qu’un jeu difficile.)
- ❖ **Binaire** : **paielement** | **mode de jeu**  
La sélection d’un choix dévalorise totalement les autres choix possibles
- ❖ **Intervalle** : **note** | **prix** | **date de sortie**  
Score maximal si le champ de l’offre appartient à l’intervalle donné, dévalorisation du score sinon selon des paramètres (un prix plus faible ne dévalorise pas, une note plus haute non plus).
- ❖ **Multiple** : **Type d’histoire**  
Les offres se rapprochant le plus de l’ensemble sélectionné par l’utilisateur auront un score élevé.
- ❖ **Cas particulier** : **genre**  
Les offres ayant un genre qui appartiennent au même groupe ou même sous-groupe de genre que le genre sélectionné par l’utilisateur auront un bonus.

## C – Implémentation des scores

Nous avons choisi de nous focaliser dans un premier temps sur l'implémentation des scores, qui représentait l'essentiel du fonctionnement de notre application. Chaque type de score possède son propre algorithme auquel on relie différents critères (voir les définitions des types de score et le projet du point de vue fonctionnel).

## D – Implémentation de la demande et de l'offre

Une fois les scores établis et fonctionnels, nous avons pu créer une offre et une demande afin de vérifier le bon fonctionnement de nos scores. Une offre possède donc en attribut chaque critère forcément rempli, alors que la demande ne possède comme critère rempli que ceux choisis par l'utilisateur.

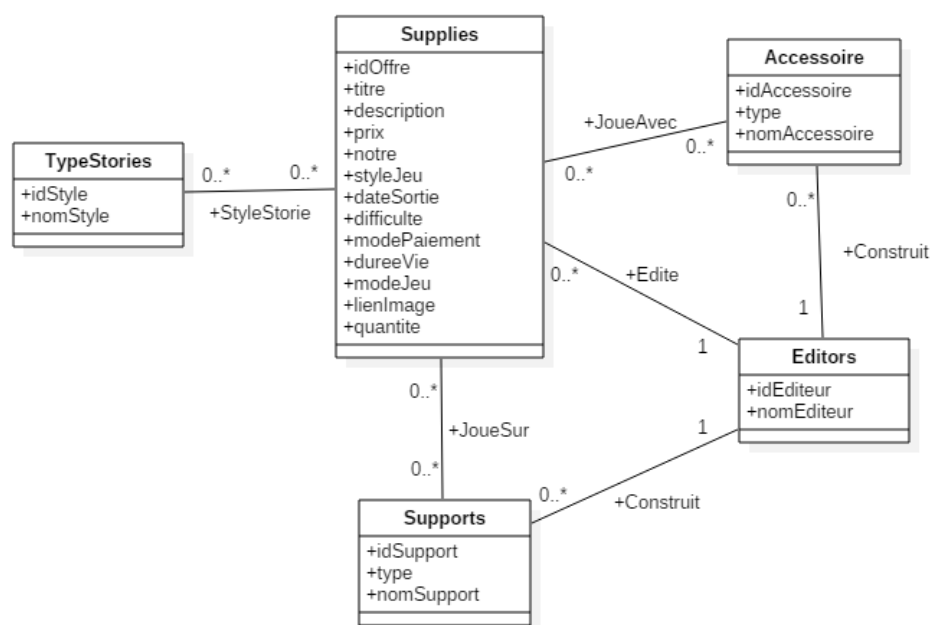
## E – Définition de la base de données

Après avoir implémenté le noyau de l'application, nous avons décidé d'implémenter une base de données. Celle-ci nous permettant de stocker l'ensemble des offres que notre catalogue propose. De plus, comme le sujet le spécifiait, un administrateur devait pouvoir administrer les offres (création, suppression, modification), donc dans une volonté de persistance des données cette option semblait être la meilleure.

### 1 – Choix du SGBD

Le choix de **SQLite** comme SGBD pour notre application, s'est fait sur son intégration au programme. Contrairement aux serveurs de bases de données traditionnelles (*MySQL*, *PostgreSQL*), SQLite est directement intégré au programme et stocké dans un fichier indépendant de la plateforme, mais aussi par son extrême légèreté.

### 2 – Modèle relationnel



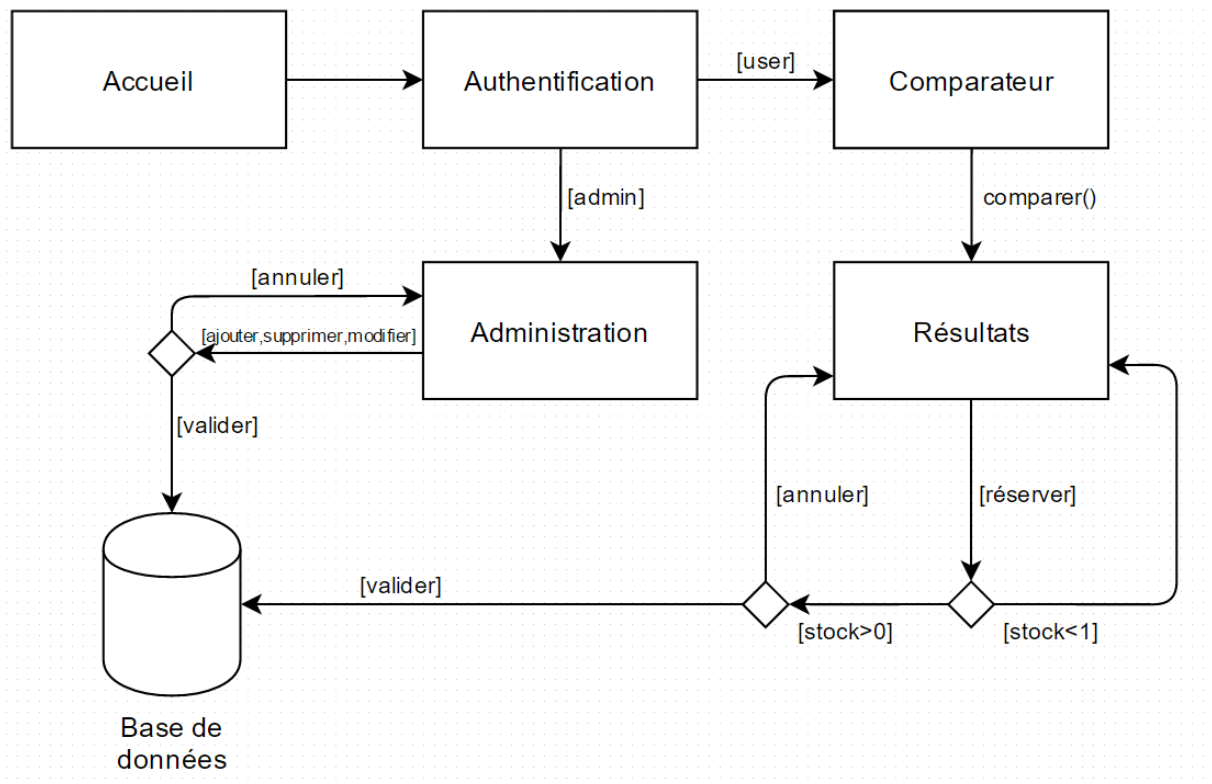
**Schéma 1 : modèle relationnel de la Base de données**

## F – Définition de l'IHM

Une IHM complète a été pensée pour compléter notre application. Celle-ci comprend une **interface d'administration** et l'**interface du comparateur**, tous deux accessibles après authentification. L'interface d'administration permet à l'utilisateur pouvant se connecter en tant qu'admin, d'ajouter, modifier ou supprimer des offres alors que l'interface du comparateur permet à l'utilisateur ayant un compte sur l'application de rentrer les critères voulus afin de comparer sa demande avec notre base de données.

Par souci de simplicité, nous avons choisi de laisser en clair les identifiants/mots de passe des sessions utilisateurs et administrateurs dans le code, et avons délibérément choisi pour l'utilisateur le nom **user** et le mot de passe **user**, et pour l'administrateur le nom **admin** et le mot de passe **admin**.

Le diagramme d'activité représentant le comportement de l'IHM est donné ci-dessous :

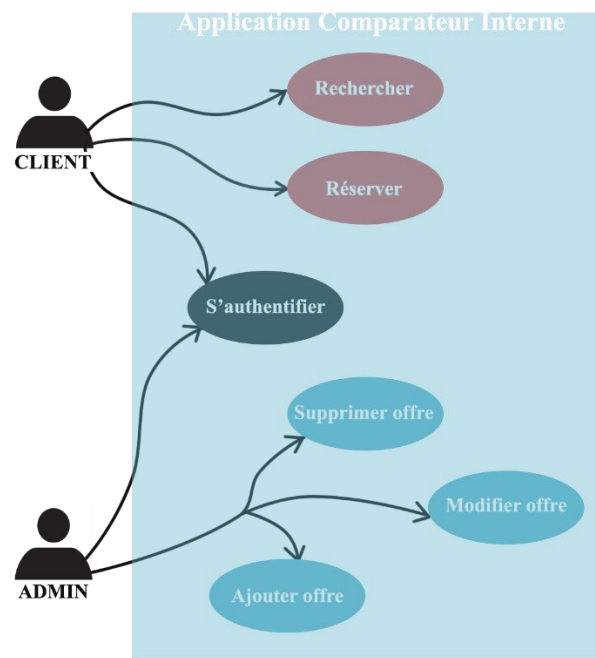


**Schéma 2 : Diagramme d'activité de l'IHM**

### III – Le projet d'un point de vue UML

#### A – Diagramme de cas d'utilisation

- ❖ L'utilisateur a la possibilité d'effectuer une recherche à partir d'une demande, puis de réserver une offre parmi les résultats retournés.
- ❖ L'administrateur a la possibilité de gérer les offres : création, modification et suppression, après authentification.



**Schéma 3 : Diagramme UML cas d'utilisation**



## B – Diagramme de séquence

### 1 - Côté client

1. Le client s'authentifie avec son mot de passe et son login.
2. L'application contrôle l'identité de l'utilisateur.
3. L'application lui affiche l'interface de recherche.
4. Il saisit les critères sur l'interface afin de trouver les jeux qui correspondent à ses attentes.
5. L'application va contacter une base de données contenant l'ensemble des jeux répertoriés.
6. La base de données lui retourne l'ensemble des données (privées de celles ne correspondant pas aux critères bloquant).
7. L'application effectue toutes les comparaisons Offre <-> Demande, en leur affectant un score.
8. On retourne au client une liste contenant un certain nombre défini de jeux classés par ordre décroissant de pertinence vis-à-vis de la demande.

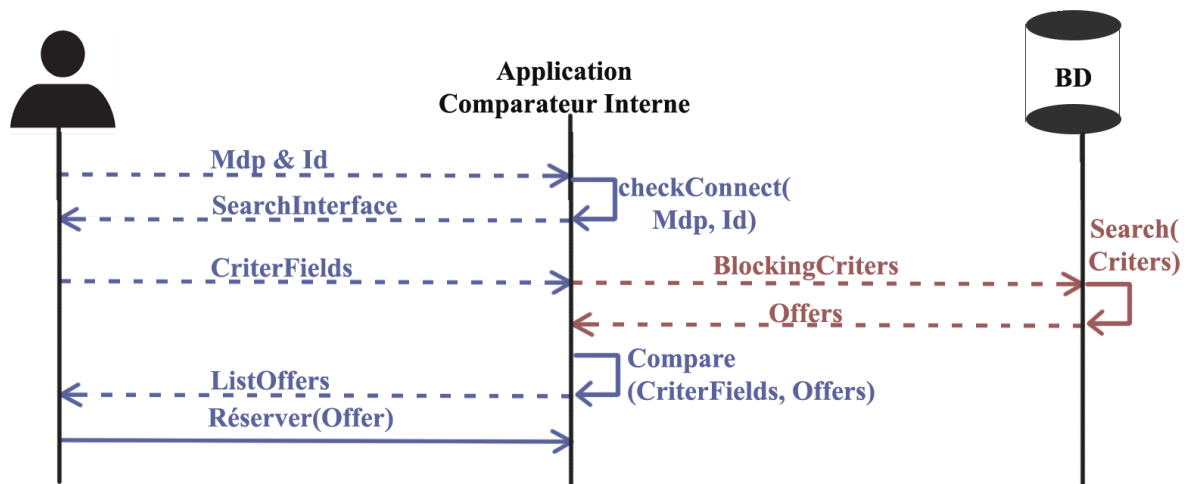
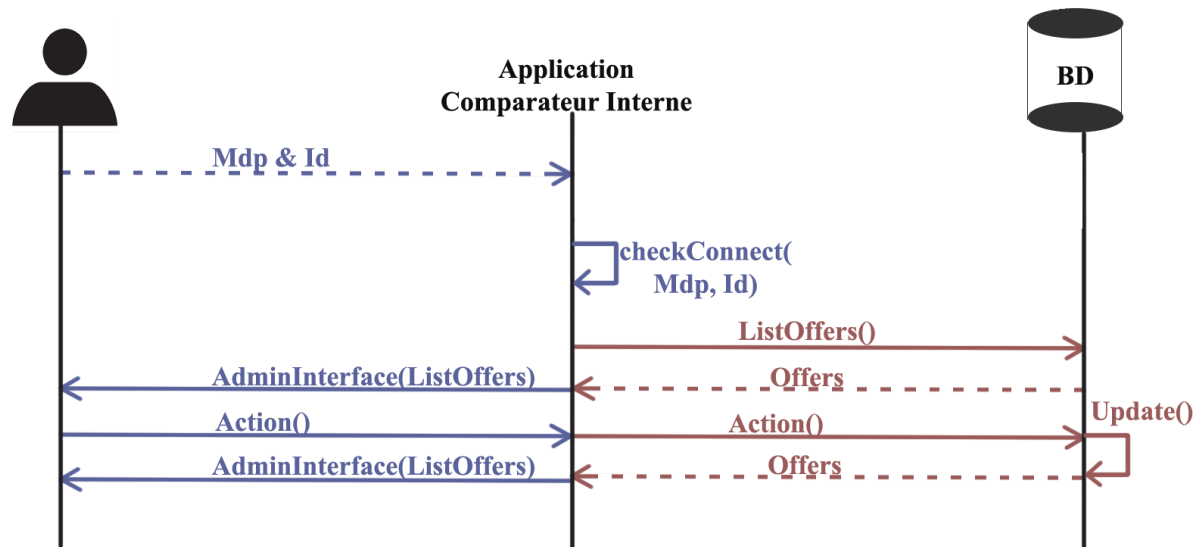


Schéma 4 : Diagramme de séquence client

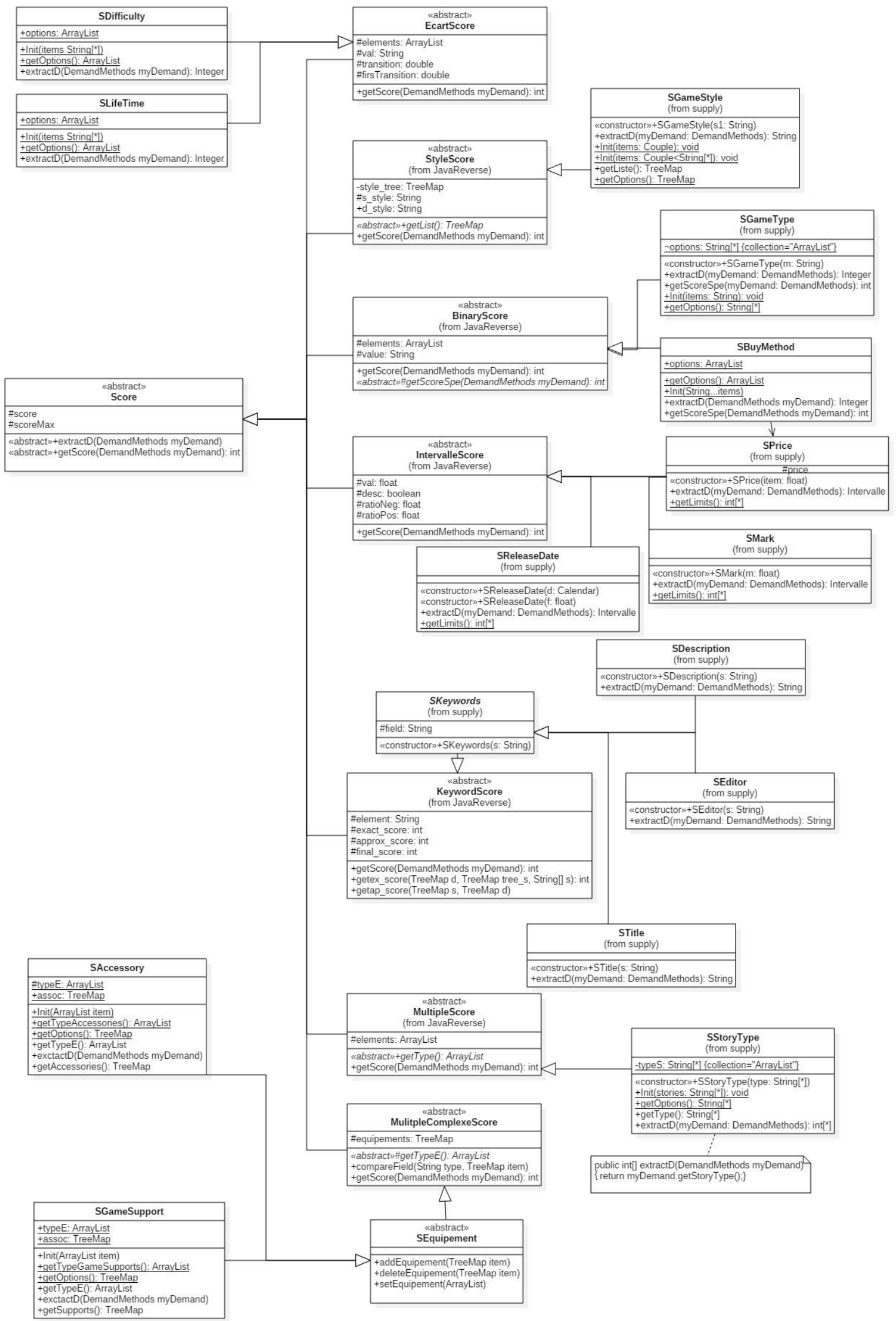
## 2 - Côté administrateur

1. Le client s'authentifie avec son mot de passe et son login.
2. L'application contrôle l'identité de l'utilisateur.
3. L'application contrôle l'authentification.
4. Elle récupère auprès de la base de données l'ensemble des offres.
5. L'application lui affiche l'interface de gestion des offres.
6. Il modifie | ajoute | supprime une offre.
7. Une fois l'action réalisée sur l'interface celle-ci sera répercuté dans la base de données.
8. Puis l'administrateur se retrouvera sur la même interface, mise à jour des modifications effectuées.

**Schéma 4 : Diagramme de séquence administrateur**

## C – Diagramme de classe perspective logicielle (noyau)

(cf. page suivante) **Schéma 5 : diagramme de classe perspective logicielle noyau**

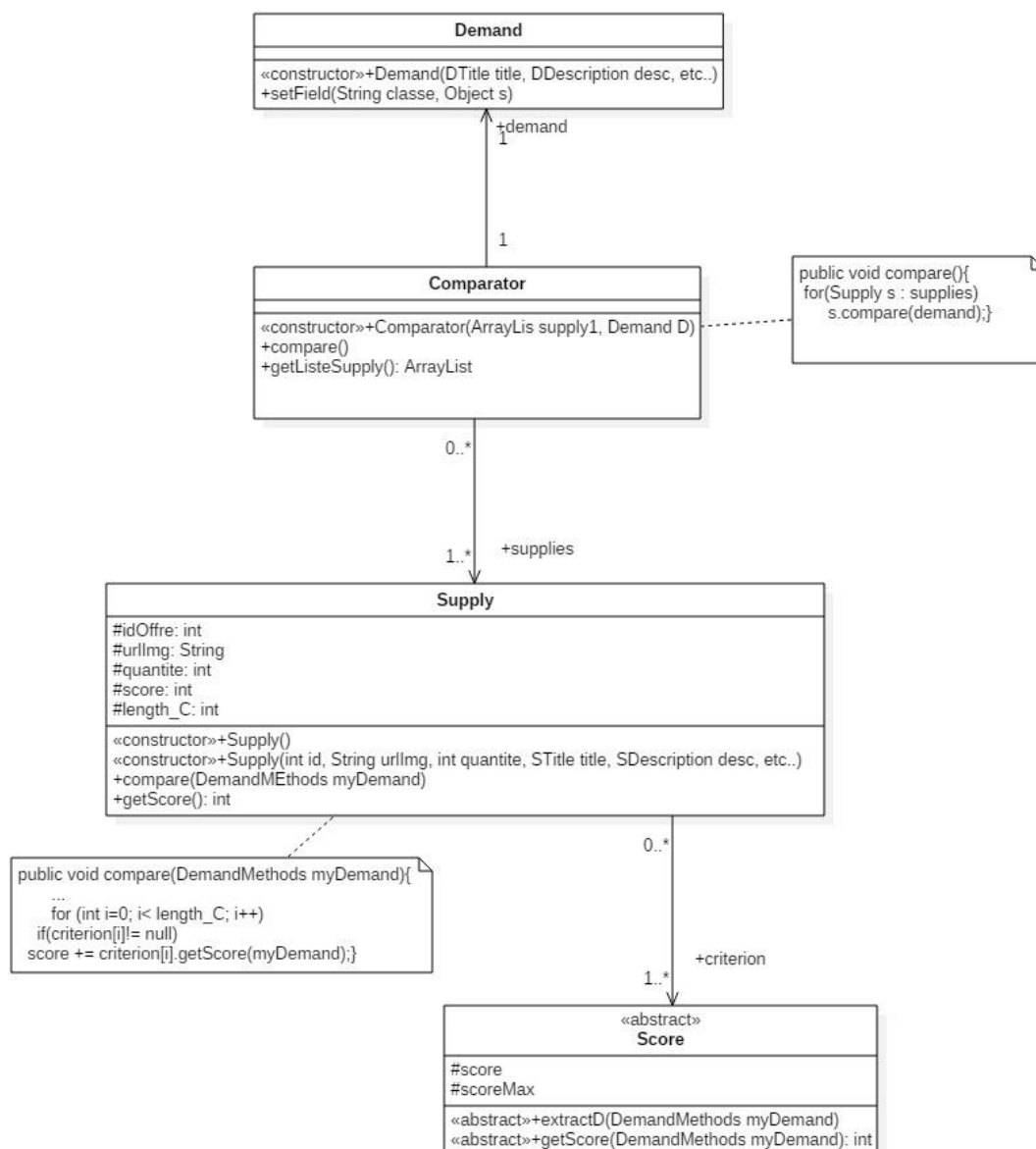


- ❖ La classe **Supply** stocke l'ensemble des critères présent dans le diagramme ci-dessus.
- ❖ La classe **Demand** comporte elle aussi l'ensemble des critères, avec les critères ayant comme nom de classe : **D[nom\_du\_critère]**. Chacune de ces classes, possède un constructeur qui dépend du type de score associé (cf. les différents scores).

Cet éclatement des critères autour des scores, permet de simplifier la programmation des classes, puisque tous les calculs sont regroupés dans les classes scores. Finalement le rôle de chaque classe critère (dans l'offre) est simplement de récupérer son homologue dans la demande et d'offrir à l'utilisateur l'ensemble des choix possibles s'il y en a **[SAccessory connaît son homologue dans demande : DAccessory, et il expose l'ensemble de ses options avec *getOptions* (utilisés pour l'IHM)]**.

Si on regarde maintenant à un plus haut niveau, en ne prenant que les classes « primaires » qui servent à la comparaison : **Comparator | Supply | Demand | Score**.

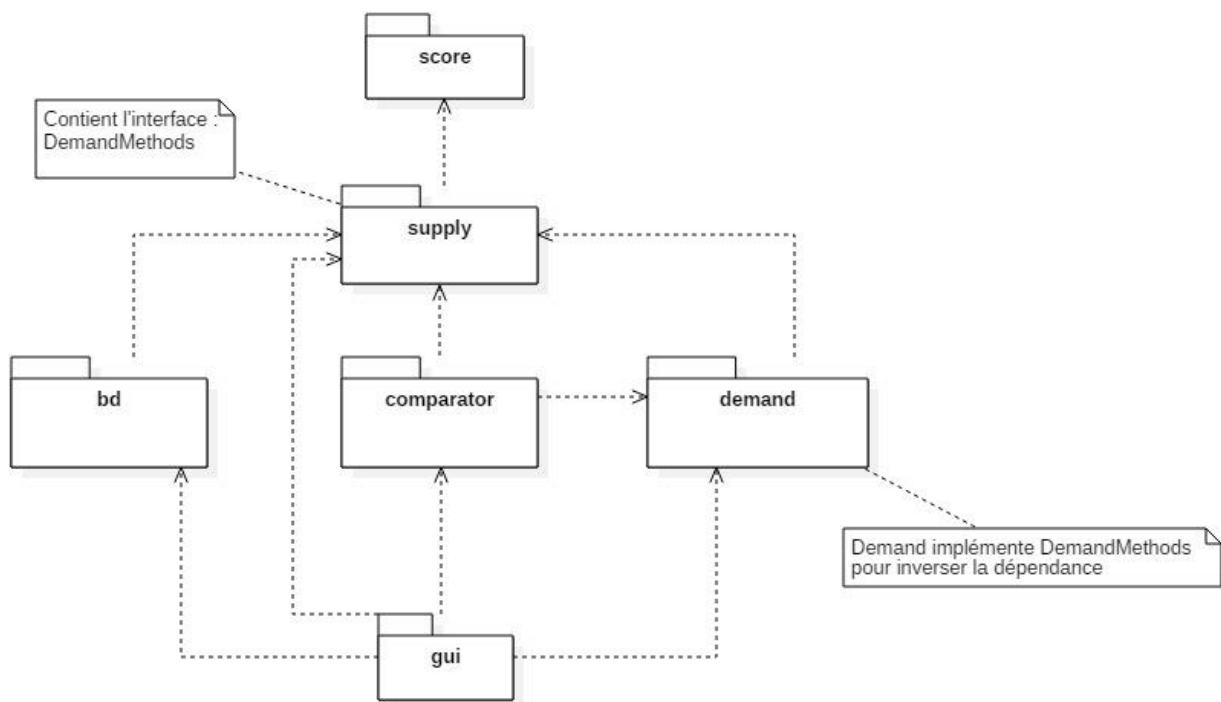
On peut voir que la comparaison de chaque critère se fait simplement en parcourant un tableau de **Score** (puisque chaque critère est affilié à un score fils de la classe mère **Score**). L'objet **Comparator**, n'a donc plus qu'à appeler pour chacune des offres, la méthode **compare()** qui va comparer l'offre à la demande.



***Schéma 6 : Diagramme de classe (perspective logicielle) des classes de haut niveau***

## D – Diagramme de package

Le diagramme de package illustre notre volonté d’avoir un code cohérent dans l’utilisation, le package **score** peut être réutilisé dans un projet équivalent mais avec d’autres critères sans souci. Le package **supply** se veut lui aussi indépendant de tout autre package que le **score**, pour cela nous avons dû inverser la dépendance entre le package **demand** et **supply**, grâce à l’interface **DemandMethods** qui regroupe l’ensemble des méthodes d’accès aux critères de la demande. Cette dépendance initiale étant due au fait que chaque critère d’une offre est le seul à connaître l’existence de son homologue dans la demande via la méthode **extractD(DemandMethods myDemand)** qui va appeler le bon getter de la demande (comme vu dans le premier diagramme de classe logicielle).



**Schéma 7 : Diagramme de package de l'application**

## E – Design Pattern

### ❖ Création dynamique de panel en fonction du critère - Design Pattern Reader

Le Panel **CriterionPanel** encapsule la création de chaque sous-Panel correspondant aux critères (il encapsule la création d'un **KeywordPanel**, d'un **EcartPanel**...). Le Design Pattern Reader permet la création de tels panels en utilisant le polymorphisme. Nous avons aussi ajouté de la réflexivité afin de rendre ce processus totalement dynamique et dépendant d'une entrée située dans le fichier config.json.

### ❖ Création des scores en fonction de types de scores et extraction du champ demandé – Design Pattern Template Method

En précisant dans la classe **Score** les méthodes **getScore()** et **extractD()**, nous avons pu créer un modèle de score afin de l'appliquer sur n'importe quel nouveau score à implémenter. Il s'agit du Design Pattern Template Method.

## III – Le projet d'un point de vue fonctionnel

### A – Les différents type de scores

Comme nous l'avons vu dans l'introduction, ou encore dans le diagramme de classe perspective logicielle, les différents critères sont regroupés par score. Nous allons donc nous intéresser aux méthodes de calculs mises en place pour chacun des scores (pour connaître les critères associés, se reporter au *schéma 5 : diagramme de classe perspective logicielle noyau*).

#### 1 – Score binaire

L'utilisateur choisit une option parmi plusieurs valeurs. L'offre obtient tous les points pour les critères associés si elle possède l'option choisie, sinon rien.

```
/**
 * Définition de la méthode héritée de Score.
 * @return score : int correspondant au score de ce champ.
 */
public int getScore(DemandMethods myDemand) {
    Integer field = extractD(myDemand);

    //Vérifie si le critère de l'offre correspond à la demande.
    return score = (val.equals(elements.get(field))) ? scoreMax+getScoreSpe(myDemand) : 0+getScoreSpe(myDemand);
}
```

Ce score prend un élément String, et récupère un int qui correspond au numéro de l'option.

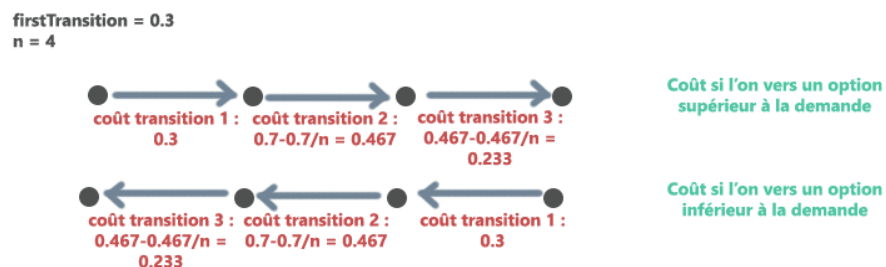
#### 2 – Score Ecart

L'utilisateur va choisir une option parmi plusieurs valeurs triées. L'offre va obtenir un score pour ce critère qui va dépendre de la distance de sa valeur par rapport à l'élément choisi dans la demande.

Pour ce score l'algorithme est plus complexe. Le but était d'avoir un algorithme qui permettait d'avoir entre les deux extrémités un coût de 100%, mais aussi de prendre en compte que le coût de la transition entre une valeur proche d'une extrémité est moins grave que la transition entre deux options au milieu.

- ❖ La transition entre l'option i-1 et i a pour coût *[Et le coût de la première transition à initialiser]*:
- ❖  $\text{reste} - \text{reste}/\text{nbIntervalles}$ .
- ❖ Où  $\text{reste} = 1 - (\text{Somme}(\text{Coût Intervalles de } 0 \text{ à } i-1))$ .

Le score de l'option i par rapport à la j sera de :  $\text{scoreMax} * (1 - (\text{Somme}(\text{coût Intervalles entre i et j})))$ .



**Schéma 8 : Exemple EcartScore**

Donc si la demande correspond à l'option 3, et que l'offre propose l'option 1 le score sera de  $\text{scoreMax} * (1 - (0.233 + 0.467))$ .

Ce score prend un élément String, et récupère un int qui correspond au numéro de l'option.

### 3 – Score Intervalle

Ici l'utilisateur va fournir un intervalle de tolérance octroyant à toutes valeurs présentes dedans, le score maximal. Ensuite plus on s'éloigne de l'une des deux bornes, plus le score diminuera. Une variable supplémentaire permettra de déterminer pour chaque critère si une valeur plus basse est moins impactante qu'une valeur plus haute ou inversement [*par exemple un prix plus faible que l'intervalle est moins problématique, qu'un prix plus haut*].



**Schéma 9 : Illustration calcul IntervalleScore**

Dans le cas où une valeur plus faible est moins grave **ratioI** sera **ratioNeg** ( $< 1$ ) et **ratioS** sera **ratioPos** ( $\geq 1$ ), dans le cas contraire cela sera l'inverse.

Ce score prend un élément *int* (l'offre), et récupère un *Intervalle* (deux *int*) de la demande.

### 4 – Score Keyword

L'algorithme utilisé pour la comparaison de **keyword** se base sur le fait que les critères utilisant ce modèle sont composés de chaînes de caractère. Nous allons créer deux scores qui seront ensuite divisés par le nombre de mots nous permettant donc d'avoir un pourcentage de proximité entre les deux chaînes.

Une première comparaison pour le score 1 est faite entre les mots du champ de type **Keyword** de l'offre et celui de la demande. On note **d\_ensemble** l'ensemble des mots de la demande et **o\_ensemble** l'ensemble des mots de l'offre.

Chaque ensemble comporte des mots séparés par un espace, on stocke donc chaque ensemble dans un tableau de chaînes de caractères :

- ❖ **d\_ensemble** : `String[]`
- ❖ **o\_ensemble** : `String[]`

On compare ensuite les mots de ces deux tableaux ; lorsqu'il y a « match » (un mot identique dans les tableaux), celui-ci est retiré du tableau de la demande et on augmente le score.

La deuxième comparaison est utilisée pour vérifier le contenu des mots : alors que la comparaison 1 regarde le mot comme une « boîte noire », la comparaison 2 regarde la composition des mots et les lettres utilisées. Pour chaque mot restant dans **d\_ensemble**, on regarde quel mot se rapproche le plus en terme de lettres de celui-ci dans **o\_ensemble** et on conserve le score associé à celui-ci. Une fois le pourcentage de chaque mot trouvé, on fait la somme de ces scores divisée par le nombre de score, ce qui nous donne un pourcentage de ressemblance générale.

## 5 – Score Multiple

Ici l'utilisateur possède un ensemble de choix, il peut en sélectionner autant qu'il le souhaite.

- ❖ On compte le nombre de choix communs entre l'offre et la demande : **cpt**.
- ❖ Le score correspond à : **scoreMax\*cpt/nbChoixDemand**

*Ce score prend un ensemble de String (l'offre), et récupère un ensemble d'int de la demande.*

## 6 – Score Multiple Complexe

Ici l'utilisateur est dans la même situation que précédemment, la différence se fait dans le calcul du score.

Ce score est construit différemment, il range les choix par catégorie *[par exemple pour les accessoires : on a manette, planche etc...]* à l'aide :

- ❖ d'un TreeMap avec la catégorie comme clé.
- ❖ en valeur un ArrayList<TreeMap<String, String>> qui prend chaque choix (la clé un descripteur *[exemple : « nomEditeur », « nomAccessoire »]*, et sa valeur).

En terme de réutilisabilité, les critères qui utilisent ce score doivent être stockés dans la base de données, et avoir une colonne du nom type, après le reste est totalement libre (totalement géré dynamiquement par la classe **Connexion**).

Donc, si l'utilisateur choisit une option qui appartient à une certaine catégorie, quand l'on va comparer si aucun élément d'une catégorie ne correspond dans l'offre, mais que l'offre possède quand même au moins un élément de cette catégorie alors on rajoutera l'équivalent de la moitié d'une correspondance.

Sinon le calcul reste le même que pour le scoreMultiple.



## 7 – Score Style

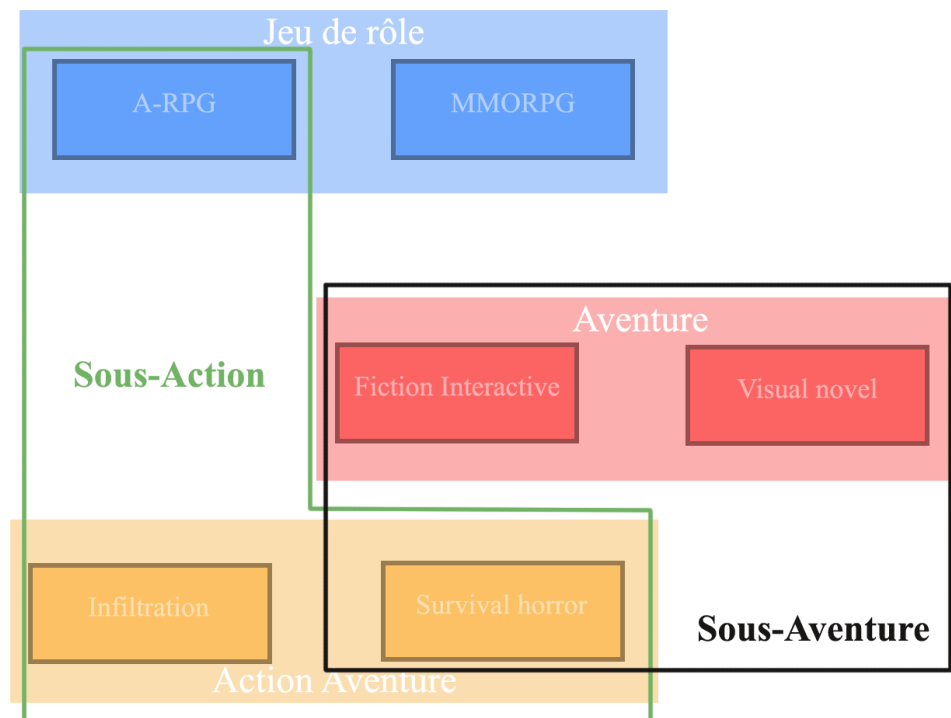
Chaque style de jeu appartient à un groupe principal de style, et un ou plusieurs groupe(s) secondaire(s).

La structure choisie pour représenter ce score est donc un **TreeMap<String,String>**,

- ❖ la clé correspondant au style de jeu
- ❖ la valeur correspondant aux groupes de styles, le premier étant le style principal.

Pour comparer deux styles entre eux, il suffit de regarder les groupes qui leur sont associés :

- ❖ s'il s'agit du **même style** (e.g. « MMORPG » et « MMORPG »), le score est maximal.
- ❖ s'il s'agit de **deux styles du même groupe principal**, le score est élevé.
- ❖ s'il s'agit de **deux styles appartenant à un même groupe secondaire**, le score est bas
- ❖ le score est nul sinon



***Schéma 10 : Illustration d'un regroupement possible pour le genre de jeu***

## B – Utilisation coté programmeur

La réutilisabilité de notre application a été la qualité principale recherchée au cours de l'implémentation de celui-ci. Le programmeur peut donc facilement modifier les informations de l'application. Toute la partie dynamique du programme est modifiable via le fichier **config.json** situé dans le dossier **src** et la **base de données : VideoComparator.db**

### 1 – Comment ajouter un critère [Réutilisabilité]

Pour créer un critère, il faut dans un premier temps créer le critère dans le **package demand** et dans le **package supply**.

#### ➤ Critère Demand

Le programmeur doit implémenter une méthode permettant de retourner le contenu de ce critère. Et ajouter sa signature dans **l'interface DemandMethods**.

#### ➤ Critère Supply

- ❖ Le programmeur doit faire hériter à sa nouvelle classe le type de score voulu, implémenter la méthode **extractD (DemandMethods myDemand)** permettant de récupérer le contenu du critère de la demande.
- ❖ Eventuellement implémenter une méthode d'initialisation des valeurs des options possibles et de récupération de ceux-ci :
  - ✓ Si le critère utilise la base de données **[Accessoires, supports, style d'histoire]** : si c'est un **MultipleComplexeScore**, alors il faut qu'il y ait au moins une colonne type dans la table et implémenter les méthodes dans la classe **Connexion**.
  - ✓ Si le critère utilise le fichier de configuration (**config.json**) **[difficulté, gameType etc..]** : il suffit de mettre une ligne : "[NOM\_CRITERE]" : ["option1", "option2", etc...]

Exemple pour BuyMethod : `46 "BuyMethod": ["Abonnement", "Licence", "Gratuit"],`

- ❖ Puis il suffit de l'ajouter à la classe principale Supply.

### 2 - Création d'un type de score

La création d'un type de score, qui est donc un ensemble de critère, est assez simple.

- ❖ Le nouveau score doit être dans le **package Score** et doit hériter de la classe Score.
- ❖ Le constructeur de la classe doit mettre en place les éventuelles constantes/limites (valeurs maximales d'un intervalle par exemple) des champs concernés.
- ❖ Le programmeur doit enfin implémenter son algorithme de calcul du score dans la méthode **public int getScore()** qui prend en argument **DemandMethods**, interface regroupant les méthodes d'accès aux critères d'une demande.

### 3 - Ajout d'un élément à l'interface du Comparateur

L'IHM utilise la réflexivité afin de pouvoir dynamiquement créer des panneaux en fonction du type de critère sélectionné.

Pour ajouter un nouveau type d'affichage à l'interface du comparateur, il suffit :

- d'ajouter son nom dans le tableau d'éléments correspondant au panneau voulu (Informations principales ou complémentaires) situé dans config.json
- d'ajouter le nouveau type de score et les classes concernées à l'intérieur d'interface selon la syntaxe suivante : (exemple pour Intervalle)

```

18     "Intervalle":
19     [
20         {"classe": "Mark", "label": "Note : ", "methods": ["getLimits"]},
21         {"classe": "Price", "label": "Prix : ", "methods": ["getLimits"]},
22         {"classe": "ReleaseDate", "label": "Sortie France : ", "methods": ["getLimits"]}
23     ],

```

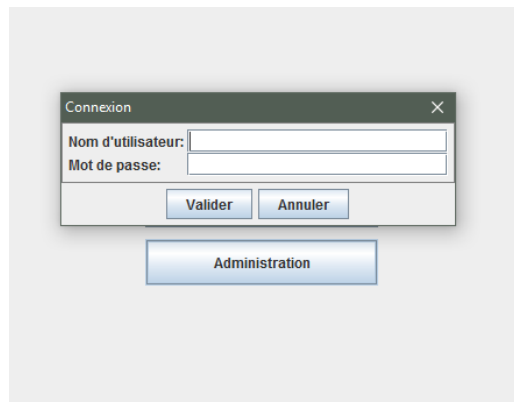
*Image 1 : Illustration de la configuration d'un type de score pour l'IHM via config.json*

## C – Utilisation coté utilisateur

### 1 – Administration des offres

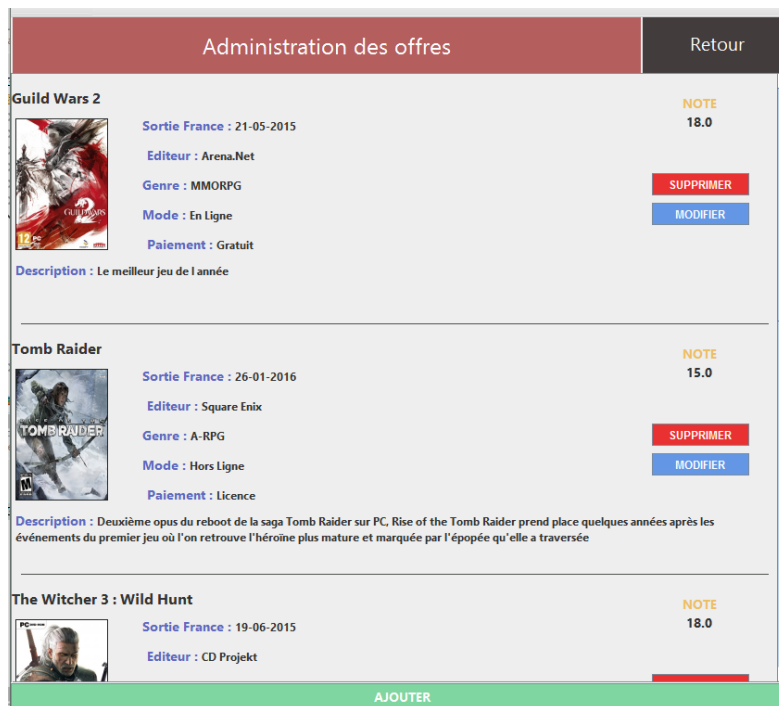
#### a – Connexion à l'interface d'administration

- ❖ Pour commencer, se connecter à l'interface d'administration en rentrant :  
**mdp : admin | password : admin.**



*Image 2 : Interface de connexion au panneau d'administration*

- ❖ Vous arrivez sur l'interface d'administration avec l'ensemble des offres disponibles dans la base de données :



**Image 3 : Interface d'administration**

b – Suppression d'une offre

- ❖ Pour supprimer une offre il suffit de cliquer sur le bouton
- ❖ Puis de confirmer la suppression via la boîte de dialogue :



La classe Connexion va effectuer la suppression dans la base de données via l'id de l'offre sélectionnée.

c – Création d'une nouvelle offre

- ❖ Pour ajouter une offre il suffit de cliquer sur :
- ❖ On remplit les champs de l'offre :



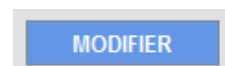
**Image 4 : Interface de d'ajout d'une offre**

- ❖ On enregistre l'offre, ce qui rafraichi le panneau d'administration avec la nouvelle offre :

**Image 5 : Visuel de l'offre crée sur l'interface d'administration**

#### d – Modification d'une offre

- ❖ Il suffit pour l'offre voulue, de cliquer sur le bouton :
- ❖ Et d'effectuer les modifications.



## 2 – Création d'une demande et réservation

- ❖ L'utilisateur doit simplement rentrer les différentes informations souhaitées pour le jeu qu'il compte réserver.

Le Hoatton - Comparateur

Fichier Aide

## Comparateur de jeux vidéos

Retour

### Informations principales

Titre du jeu :

Mots-cles :

Editeur :

Note :

Sortie France :

Genre :

Prix :

Mode de jeu : ☒ En Ligne ☐ Hors Ligne

### Informations complémentaires

Difficulte :

Duree de vie :

Accessoires : ☐ Xbox360 ☐ Volant ☐ Kinect ☐ XboxOne ☐ GamePad ☐ WiiMote

Support de jeu : ☐ 3DS ☐ PlayStation1 ☐ Apple ☐ PSVita ☐ Play Staiton4 ☐ Fixe ☐ PSP ☐ PlayStation3 ☐ Portable ☐ PlayStation2

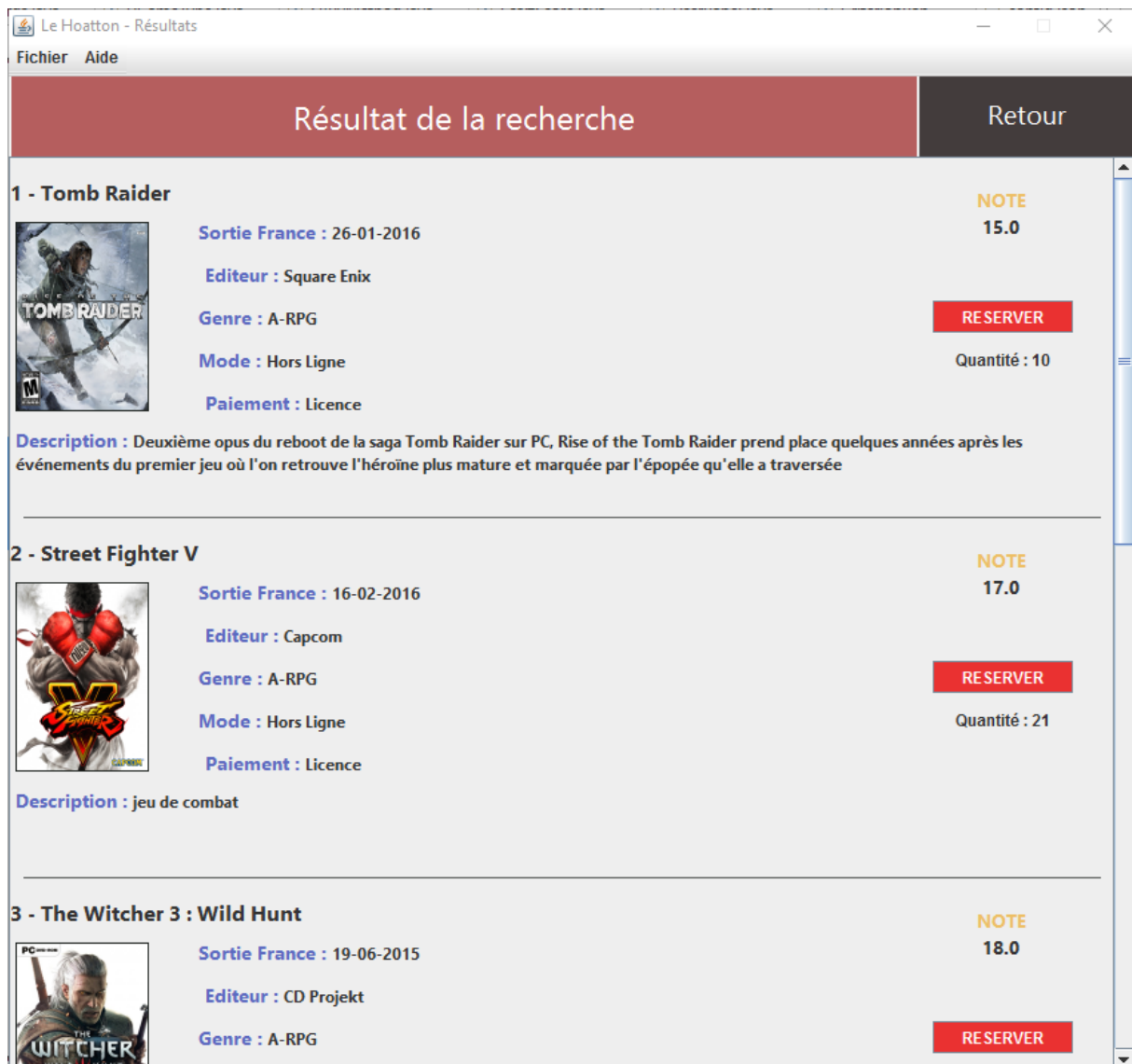
Type d'histoire : ☐ Fantastique ☐ Aventure ☐ Fiction ☐ Horreur

Paiement : ☒ Gratuit ☐ Licence ☐ Abonnement

RECHERCHER

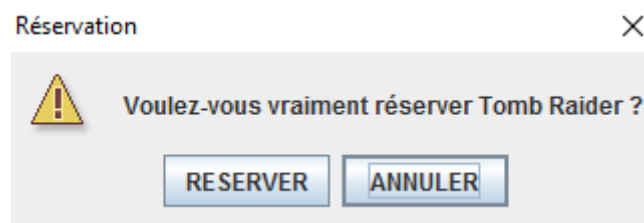
*Image 7 : Interface du comparateur*

- ❖ Il obtiendra ensuite le résultat de sa recherche classé en fonction de ses critères



*Image 8 : Interface de résultat de la comparaison*

- ❖ Il pourra donc réserver le jeu si la quantité est suffisante (dans l'exemple, on choisit le premier jeu) :



## V – Conclusion

---

Ce projet de Programmation Orientée Objet avait pour but principal de développer un noyau en Java, capable d'effectuer une recherche d'offres triées selon la proximité avec la demande d'un utilisateur. L'objectif étant de mettre en pratique les différentes notions vues lors des cours : principes, méthodes et techniques d'analyse, de modélisation, de conception et de programmation objet.

Ce projet nous a donc permis de réfléchir à la mise en place d'une application au travers de diagrammes UML tels que :

- ❖ Le diagramme de cas d'utilisation.
- ❖ Le diagramme de séquence.
- ❖ Le diagramme de classe.
- ❖ Le diagramme de package (qui nous a entre autre permis de nous rendre compte d'un problème de dépendance, et de mettre en place une inversion de dépendance à l'aide d'une interface).
- ❖ Le diagramme de classe perspective logicielle pour le noyau de l'application.
- ❖ Le diagramme d'activité pour l'IHM.

Ce projet s'est déroulé en plusieurs étapes, que l'on pourrait voir comme des couches, chaque couche étant composée d'une phase de réflexion, conception et modélisation, suivi de l'implémentation de cette partie, avec comme support les diagrammes.

Une fois le noyau réalisé, nous avons décidé d'ajouter une base de données pour avoir une persistance des données, mais aussi un support pour stocker les données d'initialisation des options de certains critères (les accessoires, les supports, les styles d'histoire).

Nous avons aussi utilisé un fichier de configuration permettant de compléter l'initialisation dynamique de contenu qui ne nous semblait pas nécessaire de stocker dans une base de données (Mode de paiements, difficultés, etc...). Tout en permettant aussi l'initialisation de l'IHM.

Ces deux éléments sont témoins d'une volonté de vouloir rendre cette application la plus réutilisable possible et ceux à tous les niveaux :

- ❖ Simplicité de création d'un critère.
- ❖ Simplicité de création d'un score.
- ❖ Aucune dépendance pour les scores implémentés : autant de choix que possibles pour les scores simples ou multiples (on peut en ajouter ou en enlever dans le fichier de config et la base de données sans avoir à modifier le code). Pour les accessoires et supports, une seule colonne indispensable.
- ❖ De même pour l'IHM.
- ❖ Possibilité de réutiliser le squelette de l'application pour un autre thème sans difficulté (soit prendre uniquement le package score, puis construire ses propres critères, ou encore prendre aussi supply et demand et modifier simplement les données de configuration).

Enfin l'IHM était pour nous une évidence pour avoir une application aboutie, puisqu'elle rendait l'application utilisable par tous, de façon simple et la plus intuitive possible.

L'interface graphique est d'un point de vue design, non aboutie, puisque le thème souhaité n'a pas pu être entièrement implémenté, de plus nous n'avons pas eu le temps de mettre en place les utilisateurs, pour créer un historique de réservation dans cette application.