**Introduction analyse**

Tout projet informatique nécessite une analyse avancée afin de définir les différentes fonctionnalités auxquelles notre système doit répondre. Pour cela, nous avons recours aux diagrammes UML (Unified Modeling Language).

En effet, ces diagrammes permettent de définir une application selon plusieurs points de vue :

* Fonctionnel : diagramme de cas d’utilisation
* Statiques : diagrammes de classes, d’objets et de structure composite
* Dynamique : diagramme de séquences, d’états, d’activité, d’interaction, de communication et de temps.
* Implémentation : diagramme de composants, de déploiement et de paquetage.

Parmi ces 13 diagrammes que comporte UML, nous utiliserons pour l’aspect fonctionnel le diagramme de cas d’utilisation. Concernant, l’aspect dynamique le diagramme d’activité et enfin le diagramme de classe, pour l’aspect statique.

# **Diagramme de cas d’utilisation**

Le diagramme de cas d’utilisation « use cases » permet la description du comportement du système du point de vue de l’utilisateur sous forme d’actions et de réactions. Un acteur représente le rôle joué par un objet (personne ou chose) qui interagit avec le système (DEBRAUNWER & VAN DER HEYDE, 2008). Un objet peut jouer plusieurs rôles vis-à-vis du système.

Nous avons donc réalisé un diagramme de cas d’utilisation comportant trois principales fonctionnalités :

* Lire les données
* Découper les données
* Afficher les données

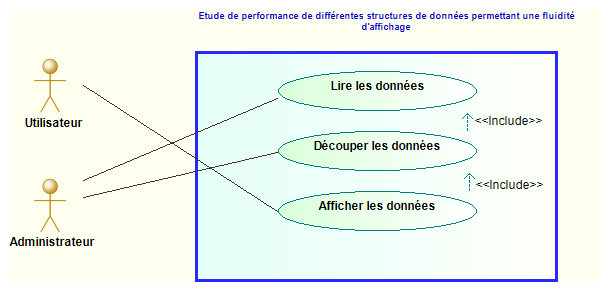


Figure 1: diagramme de cas d'utilisation

## **Fonctionnalité : Lire les données**

Cette fonctionnalité permet de *lire les données*. Elle nécessite différentes sous-fonctionnalités dont :

1. *stocker le nuage sur un support physique* : hors de notre projet

2. Ouvrir les données : opération réalisée par le système d'exploitation, hors du projet

3. *Connaître la structure du format de fichier*.

4. *Stocker en mémoire* : cela suppose de définir une structure de données adapté. En pratique, il sera impossible de stocker le nuage entier en mémoire. En dehors des métadonnées, un point est constitué, sur une machine 64 bits, de :

* indice (int)=32 bits=4 octets
* trois coordonnées : 3\*
* une coordonnée (double) = 128 bits = 16 octets soit 48 octets

Un point aura donc une taille en mémoire de 52 octets. Un nuage d'un milliard de points a donc une taille de 52 milliards d'octets soit une cinquantaine de Go, pour une mémoire vive de 4 à 8 Go), il faudra donc soit utiliser l'espace d'échange, soit ne lire qu'une partie du nuage. La structure de données devra dans tous les cas être adaptée à un grand nombre de points.

|  |  |
| --- | --- |
| **Cas d’utilisation** | **Lire les données** |
| Résumé | Ce cas d’utilisation permet de lire les données. |
| Acteur | Administrateur |
| Système | BenchOKR |
| Niveau | 1 |
| Préconditions |  |
| **Opérations** | |
| 1.1 | Stocker en mémoire les données lues |

## **Fonctionnalité : Découper les données**

|  |  |
| --- | --- |
| **Cas d’utilisation** | **Découper les données** |
| Résumé | Ce cas d’utilisation permet de construire la structure des données. |
| Acteur | Administrateur |
| Système | BenchOKR |
| Niveau | 1 |
| Préconditions | Lire les données |
| **Opérations** | |
| 2.1 | Choisir une structure |
| 2.2 | Stocker les feuilles   * 2.2.1. identifier la taille * 2.2.2. choisir le mode de stockage |
| 2.3 | Stocker la structure de l’arbre   * 2.3.1. Déterminer la hauteur de l’arbre * 2.3.2. stocker les nœuds |

## **Fonctionnalité : Découper les données**

La fonctionnalité Découper les données comprend deux sous-fonctionnalités.

|  |  |
| --- | --- |
| **Cas d’utilisation** | **Afficher les données** |
| Résumé | Ce cas permet de passer des données structurées à la représentation visuelle. |
| Acteur | Utilisateur |
| Système | BenchOKR |
| Niveau | 1 |
| Préconditions | Découper les données |
| **Opérations** | |
| 3.1 | Obtenir les points à afficher   * Requête sur un point pour obtenir un voisinage * Requête sur un volume pour obtenir les points inclus dans ce volume |
| 3.2 | Afficher les points |

# Tableau des fonctionnalités

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Fonctionnalité** | **Priorité** | **Réalisation** | **Difficulté** | **Tests** |
| 1. Lire les données | 1 | 10% | Facile |  |
| 1.1. Stocker en mémoire les données lues | 1 | 0% | Facile | Les 5 premiers points sont identiques à ceux du fichier |
| 2. Découper les données | 1 | 0% | Moyen | La structure de l’arbre est valide |
| 2.1. Choisir une structure | 1 | 0% | Facile | On peut changer la méthode utilisée |
| 2.2. Stocker les feuilles | 1 | 50% | Facile | Tous les points du nuage initial sont dans au moins une feuille. (\*) ou un échantillon représentatif … |
| 2.2.1 Identifier la taille | 1 | 0% | Moyen | Minimise l’indicateur « durée des requêtes » |
| 2.2.2 Choisir le mode de stockage | 5 | 0% | Moyen | Minimise l’indicateur « rapport taille du nuage initial / stocké » |
| 2.3 Stocker la structure de l’arbre | 1 | 0% | Moyen | Cf. infra |
| 2.3.1 Déterminer la hauteur de l’arbre | 1 | 0% | Moyen | Minimise l’indicateur « durée des requêtes » |
| 2.3.2 Stocker les nœuds | 1 | 0% | Moyen | La structure de l’arbre est vérifiée |
| 3. Afficher les données | 1 | 0% | Difficile | Cf. infra |
| 3.1 Obtenir les points à afficher | 1 | 0% | Difficile | Tous les points contenus sont récupérés |
| 3.1.1 Requête sur un point pour obtenir un voisinage | 2 | 0% | Difficile | Tous les points correspondants sont récupérés |
| 3.1.2 Requête sur un volume | 1 | 0% | Moyen | Tous les points contenus sont récupérés |
| 3.2 Afficher les points | 2 | 0% | Difficile | Les points sont affichés au bon endroit.  Les points sélectionnés sont affichés. |