Université de Versailles Saint-Quentin-en-Yvelines

UFR des sciences

Département d'Informatique

Cahier des spécifications

20 Avril 2021

Système de Gestion d'Energie pour Smart Home

Encadré par

Mme KLOUL Leila

Réalisé par

AOUCHICHE Salah

BERKANI Tinhinane

BELHANNACHI Moundji-Hocine

CHIBANI Mahmoud

CHAABANI Ahmed

HARAL Dylan

HADJ MAHFOUD Kenza

LABCHRI Koceila

2020/2021

Table des matières

Table des matières **i**

1 Introduction . . . 1
2 Modules du Projet . . . 2
3 Spécifications . . . 4
3.1 Module Objet Connecté . . . 5
3.1.1 La classe ObjetConnecte . . . 6
3.1.2 Les sous-classes de la classe ObjetConnecte . . . 9
3.1.3 La classe Routeur . . . 16
3.2 Module Calcul et Optimisation . . . 19
3.2.1 La classe Resultat . . . 20
3.2.2 La classe Calcul . . . 22
3.2.3 La classe Optimisation . . . 23
3.2.4 La classe Reglage . . . 25
3.3 Module Gestionnaire de données . . . 26
3.3.1 Choix de l'implémentation de la base de données . . . 26
3.3.2 Choix du SGBD . . . 26
3.3.3 Modèle Entité/Association . . . 27
3.3.4 La classe Connexion . . . 28
3.3.5 La classe BDDSingleton . . . 29
3.3.6 La classe Donnee . . . 31
3.4 Module Interface graphique . . . 33
3.4.1 Choix de la bibliothèque graphique . . . 33
3.4.2 La classe Main . . . 34
3.4.3 La classe GestionObjetsControlleur . . . 34
3.4.4 La classe FenetresSecondaires . . . 36
3.4.5 La classe CalculControlleur . . . 36
3.4.6 La classe UtilisateurControlleur . . . 38
3.4.7 La classe IdentificationControlleur . . . 40
3.4.8 La classe HistoriqueControlleur . . . 40
3.4.9 La classe NotificationControlleur . . . 42
4 Conclusion . . . 43
5 Glossaire . . . 43

1 Introduction

Contexte et Objectif du projet

Dans le cadre de notre module projet du semestre 6 licence informatique (LSIN608), un ensemble de projets nous a été proposés, notre choix s'est alors porté sur le thème SYSTÈME DE GESTION D'ÉNERGIE POUR SMART HOME.
Dans ce projet nous sommes chargés de développer une application qui a pour but de simuler un contrôle de la consommation énergétique d'une maison de type Smart Home, de façon à gérer la consommation et permettre ainsi à l'utilisateur de faire des économies. Notre application proposera une interface graphique sur laquelle l'utilisateur pourra suivre en temps réel sa consommation détaillée d'énergie et contrôler à distance les appareils composant sa Smart Home. Notre système va permettre de gérer plusieurs objets en même temps avec un mécanisme de file d'attente et le système sera doté d'un détecteur d'anomalies ou de surconsommations des objets connectés qui va procéder à l'optimisation en éteignant certains appareils.

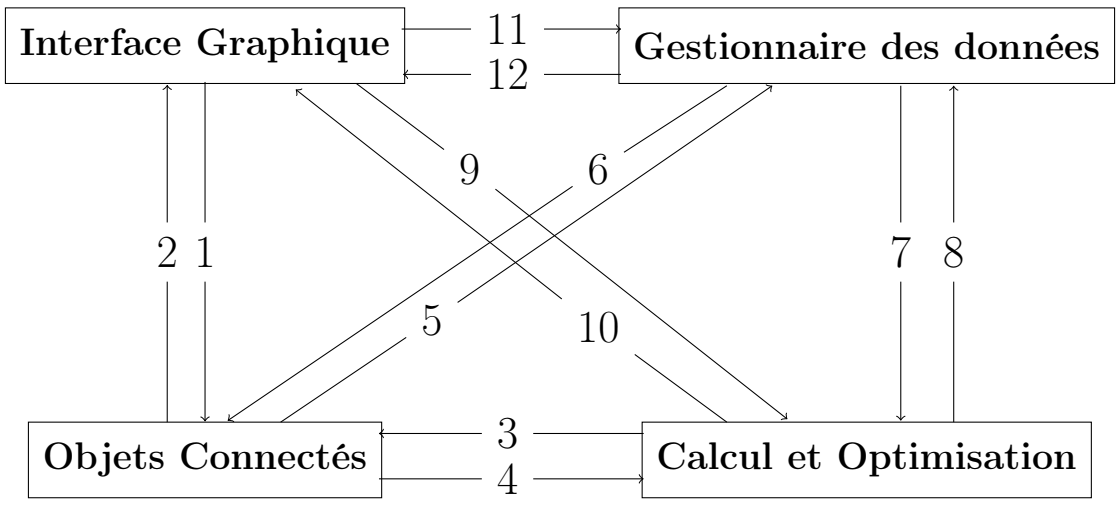
Choix du langage

Pour notre maison intelligente (SmartHome), il nous faut un langage de programmation à usage général, évolué et orienté objet. Pour des besoins spécifiques à notre projet, nous avons choisi d'implémenter notre simulateur en Java. Java possède un certain nombre de caractéristiques qui correspondent parfaitement à notre système.

- Orienté objet : dans notre application, nous manipulons plusieurs objets qui interagissent avec le système central. L'utilisation de Java va permettre l'interaction entres nos objets.
- Mécanisme de l'héritage : les objets connectés que l'on va manipuler ont tous des caractéristiques en commun, regrouper l'ensemble des caractéristiques communes dans une même classe mère est un avantage conséquent. De ce fait, la notion d'héritage présente dans le langage est importante pour la réalisation de notre projet.
- Mécanisme du ramasse-miettes : cet élément va contribuer à la robustesse et à la performance de notre système, car il récupère automatiquement la mémoire inutilisée grâce à son garbage collector qui restitue les zones de mémoire laissées libres suite à la destruction des objets.
- Gestion de multitâche : puisque notre système est composé des objets intelligents et autonomes, nous avons besoin d'un langage qui permet l'utilisation de threads qui sont des unités d'exécutions isolées.
- Portabilité de notre système : nous voulons que notre système soit indépendant de la machine sur laquelle il s'exécute. Ceci est possible grâce à la machine virtuelle de java.

2 Modules du Projet

Organigramme :



Informations qui circulent entre les modules :

1. Consultation et modification de l'état d'un objet (envoie d'ordre).
2. Mise à jour de l'état de l'objet sur l'interface.
3. Envoie de nouveaux réglages ou d'un nouvel état provenant d'un objet faisant suite à une solution d'optimisation.
4. Envoie des alertes de dépassement de seuil de consommation ou d'informations sur l'environnement de l'objet susceptibles de modifier l'état de l'objet, vers le module de calcul et d'optimisation (signaux/données).
5. Envoie de nouvelles données des objets vers le gestionnaire des données dans le cas d'une mise à jour effectuée sur l'état de l'objet.
6. Récupération des données de saisie pour effectuer l'intégration virtuelle d'un objet physique dans le système de gestion (donnée de saisie + identifiant numérique).
7. Envoie des données de référence (HP, HC, ...) et les données des objets (état, consommation, ...) pour effectuer un calcul ou une optimisation.
8. Envoie d'une requête pour les données afin d'effectuer un calcul spécifique (identifiant numérique nécessaire au calcul).
9. Envoie d'une requête pour l'opération à réaliser.
10. Transmission du résultat du calcul des données ou envoie d'une requête d'autorisation ou de confirmation à l'utilisateur (notification/signal).
11. Importation des données des objets ou envoie de requêtes d'ajouts directement à partir de l'interface.
12. Envoie d'un signal d'erreur vers l'interface graphique dans le cas d'une saisie erronée.

Fonctionnalités des modules :

Interface Graphique : Ce module permettra à l'utilisateur de contrôler son système de gestion et d'interagir avec lui.

(1)-Afficher la liste d'objets connectés qui constituent la smart home sur l'interface.
(2)-Commande de sauvegarde des informations contenues dans nos objets.
(3)-Option "Paramètres" qui permet l'application de nouveaux réglages sur les objets.
(4)-Option d'ajout d'un objet connecté au système (avec une configuration initiale à spécifier au moment de la création).

(5)-Donner une vision globale/détaillée sur la consommation énergétique de la maison (historique de consommation).
(6)-Fenêtre de visualisation de la smart home.
(7)-Fenêtre de visualisation des notifications et des traitements du système de gestion.
(8)-Afficher les différents états des objets connectés (allumé/éteint).
(9)-Afficher des boîtes de dialogue représentant les requêtes d'optimisation et qui demandent la confirmation de l'utilisateur pour certaines décisions.
(10)-Proposer une option permettant de lancer un ensemble de tests prédéfinis sur nos différents objets, afin de mettre en évidence les décisions de notre système de gestion.
(11)-Permettre de fixer le taux de consommation (économique, modéré, haute performances), ou bien de spécifier directement la consommation par watts/heures voulue.
(12)-Quitter l'application.
(13)-Option d'aide et d'assistance.
(14)-Sauvegarde de fichiers/entrées au niveau interface pour transférer au gestionnaire de données.
(15)-Proposer les solutions d'optimisation trouvées à l'utilisateur.
(16)-Minuteur/Date de l'application.

Gestionnaire de Données : Ce module permettra la sauvegarde des données qui circulent dans l'application.

(1)-Analyser et interpréter des saisies de données de l'utilisateur.
(2)-Sauvegarder et mettre à jour les données de consommation/états des objets connectés.
(3)-Envoie des données nécessaires au module calcul et optimisation pour effectuer les opérations nécessaires.
(4)-Intégration de nouveaux objets connectés au niveau de la base de données suite à la saisie de l'utilisateur.
(5)-Système de génération et mise à jour de fichiers de sauvegarde pour l'historique.

Objets connectés : Ce module représentera tous les objets connectés de la smart home.

(1)-Centraliser la communication inter-objets et objets-système grâce à un routeur représenté comme un objet connecté.
(2)-Appliquer un changement d'état suivant les ordres reçus par l'utilisateur ou par le système de gestion d'énergie.
(3)-Capter des données intéressantes pour l'utilisateur susceptibles de faire changer l'état d'un objet (température, présence d'une personne, détecteur de fumée, ...).
(4)-Transmettre les données de consommation et les états de l'objet au gestionnaire de données.
(5)-Déclenchement d'une action en fonction des informations captées. Exemple : déclenchement d'une alarme en cas d'intrusion captée par le capteur de l'alarme.

Optimisation et Calculs : Ce module permettra d'effectuer l'optimisation d'énergie et les calculs de consommation.

(1)-Calculer la consommation d'un ou plusieurs objets en se basant sur les données récoltées.
(2)-Envoyer l'ordre de changement d'état d'un ou plusieurs objets connectés.
(3)-Comparer les résultats des calculs et les données de saisie fournies par le module gestionnaire de données.
(4)-Trouver une solution optimale en fonction du résultat des différentes comparaisons faites auparavant et l'envoyer sur l'interface.

(5)-Appliquer directement la solution trouvée en envoyant de nouveaux réglages aux objets connectés.
(6)-Envoie d'une requête à la base de donnée pour avoir accès aux données nécessaires afin d'effectuer un calcul spécifique.

3 Spécifications

Les principales classes de notre application sont représentées avec le diagramme de classe suivant :

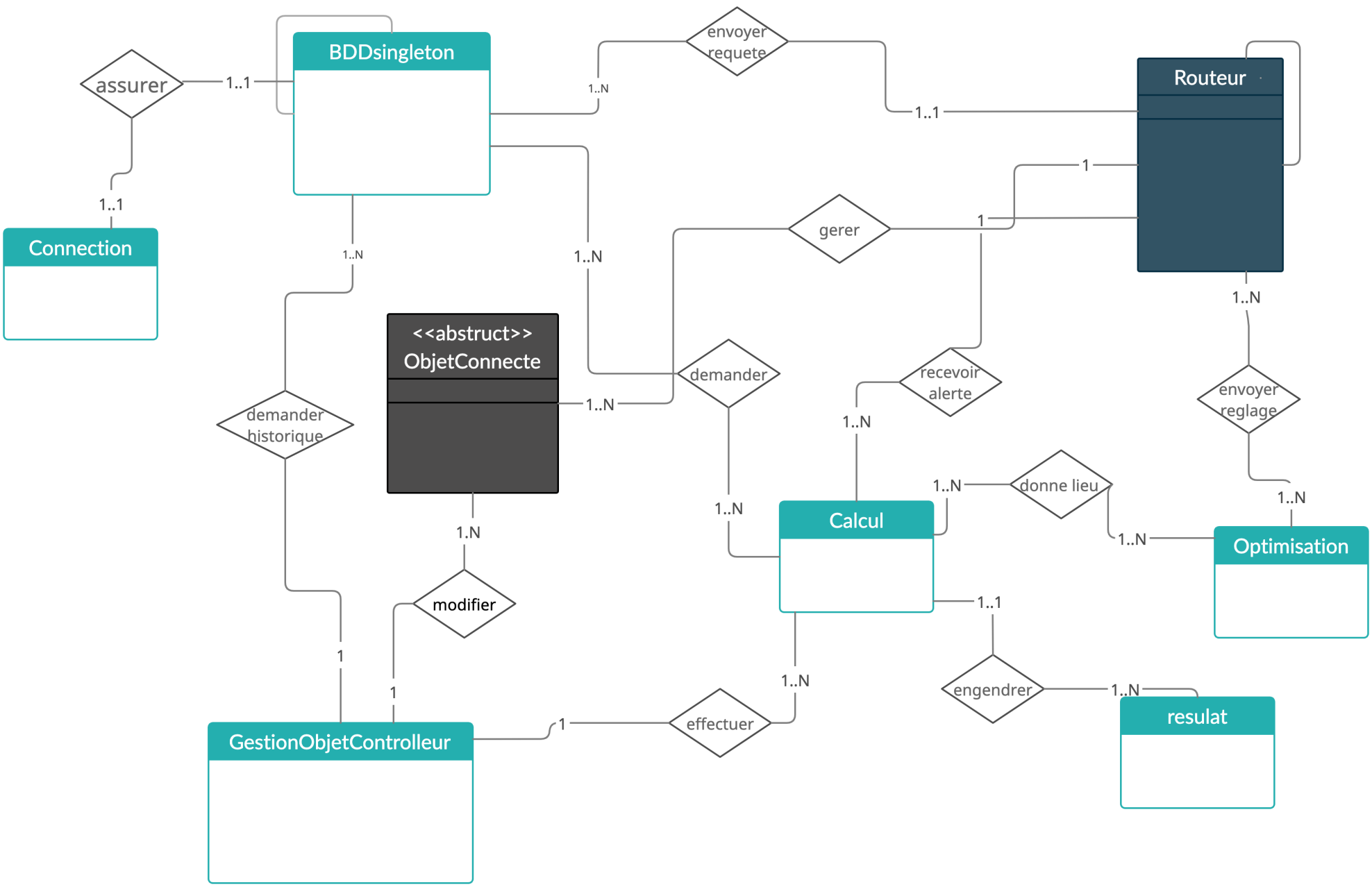


FIGURE 1 – Diagramme de classe globale

La classe ObjetConnecte appartient au module Objet Connecté et est représentée ici :
→ La classe ObjetConnecte
La classe Routeur appartient au module Objet Connecté et est représentée ici :
→ La classe Routeur
La classe Calcul appartient au module Calcul et Optimisation et est représentée ici :
→ La classe Calcul
La classe Optimisation appartient au module Calcul et Optimisation et est représentée ici :
→ La classe Optimisation
La classe Resultat appartient au module Calcul et Optimisation et est représentée ici :
→ La classe Resultat
La classe Connexion appartient au module Gestionnaire 25 de Données et est représentée ici :
→ La classe Connexion
La classe BDDSingleton appartient au module Gestionnaire de données et est représentée ici :
→ La classe BDDSingleton
La classe GestionObjetsControlleur appartient au module Interface graphique et est représentée ici :
→ La classe GestionObjetsControlleur

3.1 Module Objet Connecté

Ce module représente l'ensemble des objets connectés qui constituent la Smart Home.Nous avons choisi la liste d'objets suivantes : Chauffage, Thermostat, Lampe, Réfrigérateur, Climatiseur, Enceinte Connectée, Télévision.
Le module Objet Connecté est représenté par le diagramme des classes UML suivant :

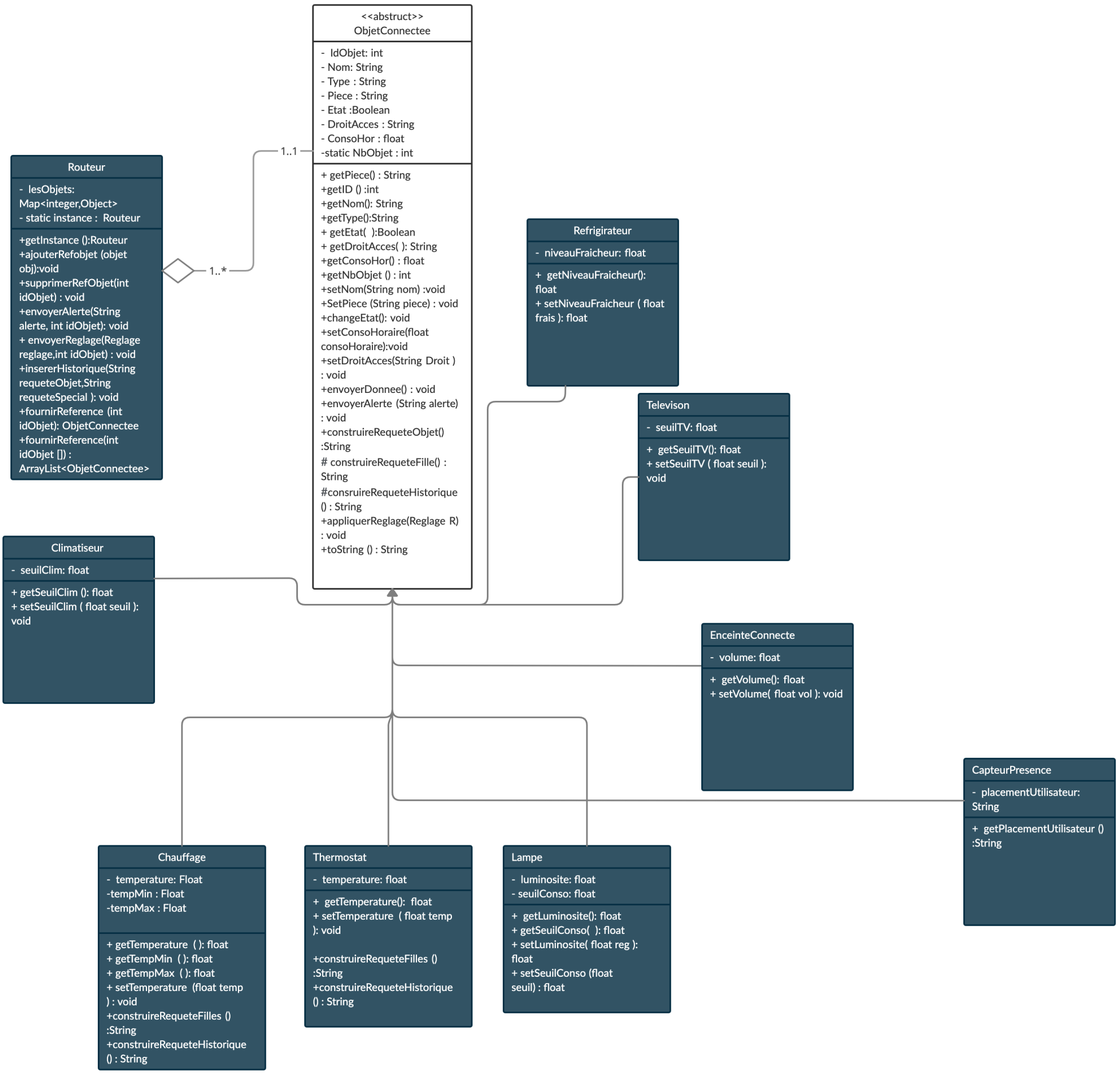


FIGURE 2 – Diagramme de classe globale

3.1.1 La classe ObjetConnecte

Cette classe est une classe abstraite qui définit les caractéristiques communes de tous les objets connectés appartenant à la Smart Home. La modification d'un objet depuis l'interface par l'utilisateur, engendre l'ajout de nouvelles données dans l'historique de l'objet modifié.

```
Public Abstract Class ObjetConnecte {
protected final int idObjet ;
protected String nom ;
protected final String type ;
protected String piece ;
protected boolean etat ;
protected String droitAcces ;
protected float consoHoraire ;
protected static int nbObjets ;

public ObjetConnecte(String nom, String type, String piece, float consoHoraire ,String
droitAccess) ;
public ObjetConnecte(ObjetConnecte Obj) ;
public int getId() ;
public String getNom() ;
public String getType() ;
public String getPiece() ;
public boolean getEtat() ;
public String getDroitAcces ;
public float getConsoHoraire ;
public static int getNbObjet() ;
public void setNom(String nom) ;
public void setPiece(String nomPiece) ;
public void setConsoHoraire(float consoHoraire) ;
public void changeEtat() ;
public void setDroitAcces() ;

public void envoyerDonnee() ;
public void envoyerAlerte(String alerte) ;
public String construireRequeteObjet() ;
public abstract String construireRequeteFille() ;
public abstract String constuireRequeteHistorique() ;
public void appliquerReglage(Reglage r) ;

public String toString() ;
}
```

Explication des attributs

idObjet : attribut de type int qui représente l'identifiant de l'objet.

nom : attribut de type String qui représente le nom de l'objet.

type : attribut de type String qui représente le type de l'objet(Lampe,Chauffage,...).

piece : attribut de type String qui représente le nom de la pièce où se situe l'objet dans la Smart Home.

etat : attribut de type boolean qui représente l'état actuel de l'objet. Il sera égal à true (Vrai) si l'objet est allumé et false(Faux) pour un objet éteint.

droitAcces : attribut de type String qui représente le droit d'accès du système à la manipulation de l'objet.Il est utile dans le cas où l'utilisateur ne veut pas que le système de gestion d'énergie modifie l'état d'un objet quelconque.

consoHoraire : attribut de type float qui représente la consommation horaire d'un objet connecté en Watt/heure.

nbObjets : attribut de type static int qui représente le nombre total d'objets qui constituent la Smart Home. Cet attribut est incrémenté à chaque instanciation d'un objet.

Explication des méthodes

public ObjetConnecte(String name, String type, String piece, float consoHoraire ,String droitAccess) : constructeur de la classe Objet, il prends en argument un identifiant, un nom, un type, un état, une piéce et un droit d'accès. Cette classe étant abstraite on ne pourra pas l'instancier.

public ObjetConnecte(ObjetConnecte obj) : constructeur par recopie de la classe "ObjetConnecte", il a un seul paramètre de même type que l'objet à créer qui est de type "ObjetConnecte", il recopie les paramètres depuis l'objet passé en paramètre sur l'objet à créer.

public String getId() : méthode qui permet de récupérer l'identifiant d'un objet.

public String getNom() : méthode qui permet de récupérer le nom de l'objet.

public String getType() : méthode qui permet de récupérer le type de l'objet.

public String getPiece() : méthode qui permet de récupérer le nom de la pièce dans laquelle se situe l'objet connecté dans la maison.

public String getEtat() : méthode qui permet de récupérer l'état de l'objet connecté.

public String getDroitAcces() : méthode qui permet de récupérer le droit d'accès du système à l'objet connecté pour modification.

public String getConsoHoraire() : méthode qui permet de récupérer la consommation horaire de l'objet en Watt/heure.

public static int getNbObjet() : méthode qui permet de récupérer le nombre d'objets qui constituent la Smart Home.

public void setNom(String nom) : méthode qui permet de modifier le nom de l'objet.Elle sera utilisée dans envoyerDonnee() pour envoyer les modifications au gestionnaire de données

public void setPiece(String piece) : méthode qui permet de modifier le nom de la pièce où se

trouve l'objet.

public void setConsoHoraire(float consoHoraire) : méthode qui permet de modifier la consommation horaire de l'objet par "consoHoraire" passée en paramètre.

public void changeEtat() : méthode qui permet de modifier l'état de l'objet.

public void setDroitAcces() : méthode qui permet de modifier les droits d'accès du système à l'objet connecté pour modification.

public void envoyerDonnee() : méthode qui permet d'envoyer toutes les informations de l'objet dans le module gestionnaire de données, en utilisant la méthode du routeur *envoyerRequete(String requete)*. Elle correspond à la fonctionnalité (4) du module objet connecté : "Transmettre les données de consommation et les états des objets au module gestionnaire de données".

public void envoyerAlerte(String Alerte) : méthode qui permet d'envoyer une alerte au module calcul et optimisation pour l'informer d'un éventuel dépassement de seuil de consommation.

public void appliquerReglage(Reglage r) : méthode qui prend en argument un objet de type Réglage (défini dans le module calcul et optimisation) et qui permet d'appliquer des réglages sur l'objet connecté concerné (après la réception d'ordre de la part du module calcul et optimisation à travers le routeur). Elle effectue donc la fonctionnalité (2) du module Objet Connecté : "Appliquer un changement d'état suivant un ordre reçu par l'utilisateur ou par le système de gestion d'énergie".

public String construireRequeteObjet) : cette méthode permet de construire la requête d'insertion afin d'intégrer l'objet dans notre base de données. Exemple : insert into ObjetConnecte Values(attributs de la classe objetConnecte). Elle se déclenche lors de l'instanciation d'un Objet

public abstract String construireRequeteFille cette méthode abstraite est à redéfinir dans les classes filles, elle permet de construire la requête d'insertion dans la base de données propre à une sous classe.Elle se déclenche lors de l'instanciation d'un Objet.

public abstract void construireRequeteHistorique() cette méthode abstraite est à redéfinir dans les classes filles, elle permet de construire une requête d'insertion de ligne d'historique dans la base de données dans la table HistoriqueObjet. Elle se déclenche lorsqu'on change un des attributs d'un objet. Ceci permet d'avoir une table d'historique unique et fonctionnelle.

String toString() : Méthode qui met sous la forme d'un String les informations de la classe.

3.1.2 Les sous-classes de la classe ObjetConnecte

Ces classes sont des sous-classes ou des classes filles de la classe ObjetConnecte, elles **héritent** de toutes les caractéristiques de la classe ObjetConnecte.Elles devront redéfinir toutes nos méthodes abstraites présentes dans la classe ObjetConnecte.
Nous avons décidé d'inclure cette liste d'objets connectés dans notre application :
(1)-Chauffage
(2)-Thermostat
(3)-Lampe
(4)-Refrigerateur
(5)-Television
(6)-Climatiseur
(7)-Enceinte Connectée
(8)-CapteurPresence

La Classe Chauffage : La Classe Chauffage hérite de tout les attributs de la classe mère ObjetConnecte et contient en plus les attributs et méthodes suivantes :

```
Public Class Chauffage extends ObjetConnecte {
private float temperature;
private final float tempMin=7.0;
private final float tempMax=35.0;

public Chauffage( String nom, String type, String piece, float consoHoraire,String DroitAcces, float température);

public Chauffage( Chauffage chauf);

public float getTemperature();
public float getTempMax();
public float getTempMin();
public float setTemperature();
public String construireRequeteFille();
public String construireRequeteHistorique();
}
```

Explication des attributs

temperature : attribut de type float qui représente la température qu'affiche le chauffage.

tempMin : attribut de type float qui représente la température minimale à laquelle l'utilisateur peut régler le chauffage. Il est "final" parce que l'utilisateur ne doit pas régler la température de son chauffage au dessous d'une certaine température minimale fixée par le fabricant du chauffage.

tempMax : attribut de type float qui représente la température maximale à laquelle l'utilisateur peut régler le chauffage. Il est "final" parce que l'utilisateur ne doit pas régler la température de son chauffage au dessus d'une certaine température maximal fixée par le fabricant du chauffage.

Explication des méthodes

public Chauffage(String nom, String type, String piece, float consoHoraire,String DroitAcces, float température) : constructeur de la classe Chauffage. Il permet la création des objets de

type "Chauffage", il prend un argument de plus que le constructeur de la classe mère ObjetConnecte qui est la température. L'état de l'objet sera mit à "faux" donc sera "éteint" par défaut, et l'identifiant de l'objet est attribué automatiquement suivant les identifiant déjà pris.

public Chauffage(Chauffage chauf) : constructeur par recopie de la classe "Chauffage", il a un seul paramètre "chauf" de même type que l'objet à créer qui est de type "Chauffage", il recopie les paramètres depuis l'objet passé en paramètre sur l'objet à créer.

public float getTemperature() : récupère la température que le chauffage affiche.

public float getTemMax() : récupère la température maximale à laquelle l'utilisateur peut régler le chauffage.

public float getTempMin() : récupère la température minimale à laquelle l'utilisateur peut régler le chauffage.

public float setTemperature() : récupère la température à laquelle le chauffage est réglé.

public String construireRequeteFille() : c'est la redéfinition de la méthode "construireRequeteFille()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

public String constuireRequeteHistorique() : c'est la redéfinition de la méthode "constuireRequeteHistorique()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

La Classe Thermostat :

La Classe Thermostat représente le capteur de température de la maison, elle hérite de tout les attributs de la classe mère ObjetConnecte et contient en plus les attributs et méthodes suivantes :

```
Public class Thermostat extends ObjetConnecte {
private float température;

public Thermostat(String nom, String type, String DroitAcces, boolean etat, float temperature);
public Thermostat(Thermostat ther);
public float getTemperature();
public void setTemperature(float);
public String construireRequeteFille();
public String constuireRequeteHistorique();
}
```

Explication des attributs

température : Attribut de type float qui représente la température qu'affiche le thermostat.

Explication des méthodes

public Thermostat(String nom,String type,String piece,float consoHoraire,String DroitAcces,float temperature) :constructeur de la classe Thermostat. Il permet la création des objets de type "Thermostat", il prend un argument de plus que le constructeur de la classe mère ObjetConnecte qui est la température. L'état de l'objet sera mit à "faux" donc sera "éteint" par défaut, et l'identifiant de l'objet est attribué automatiquement suivant les identifiant déjà pris.

public Thermostat(Thermostat ther) : constructeur par recopie de la classe "Thermostat", il a un seul paramètre "ther" de même type que l'objet à créer qui est de type "Thermostat", il recopie les attributs depuis l'objet passé en paramètre sur l'objet à créer.

public float getTemperature() : méthode qui permet de récupérer la température qu'affiche le thermostat.

public void setTemperature(float temp) : méthode qui prend en argument un float et qui permet de modifier la température qu'affiche le thermostat. Dans le cas ou la température est supérieur ou inférieur aux seuils donnés la modification n'opère pas. De plus on doit utiliser la méthode envoyerAlerte(String alerte) pour communiquer le changement au module calcul et optimisation en passant par le Routeur en utilisant la méthode envoyerAlerte(String alerte,int idObjet) de la classe routeur.

public String construireRequeteFille() : c'est la redéfinition de la méthode "construireRequeteFille()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

public String constuireRequeteHistorique() : c'est la redéfinition de la méthode "constuireRequeteHistorique()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

La Classe Lampe :

La Classe Lampe représente les différentes lampes de la Smart Home, elle hérite de tout les attributs de la classe mère ObjetConnecte et contient en plus les attributs et méthodes suivantes :

```
Public class Lampe extends ObjetConnecte{
private float seuilConso ;
private float luminosité ;

public Lampe(String nom, String type, String piece, float consoHoraire,StringDroitAcces, float
luminosité,float seuilConso) ;

public Lampe(Lampe lampe) ;

public float getLuminosité() ;
public float getSeuilConso() ;
public void setLuminosité(float) ;
public void setSeuilConso(float) ;
public String construireRequeteFille() ;
public String construireRequeteHistorique() ;
}
```

Explication des attributs

luminosité : Attribut de type float qui représente la luminosité de la lampe.

seuilConso : Attribut de type float qui représente le seuil de consommation d'une lampe.

Explication des méthodes

public Lampe(String nom, String type, String piece, float consoHoraire,StringDroitAcces, float luminosité,float seuilConso) ;
: Constructeur de la classe Lampe.Il permet la création des objets de type "Lampe",il prend deux ar-

guments de plus que le constructeur de la classe mère ObjetConnecte qui est la luminosité de la lampe et le seuil de consommation maximum autorisé de la lampe .L'état de l'objet sera mit à "faux" donc sera "éteint" par défaut, et l'identifiant de l'objet est attribué automatiquement suivant les identifiant déjà pris.
public Lampe(Lampe lampe) : Constructeur par recopie de la classe "Lampe", il a un seul paramètre "lampe" de même type que l'objet à créer qui est de type "Lampe", il recopie les attributs de l'objet passé en paramètre et il les affecte à l'objet à créer.

public float getLuminosité() : méthode qui permet de récupérer le niveau de luminosité d'une lampe.

public float getSeuilConso() : méthode qui permet de récupérer le seuil de consommation d'une lampe.

public void setLuminosité(float) : méthode qui prend un argument un float et qui permet de modifier la luminosité de la lampe.

public void setSeuilConso(float) : méthode qui prend en argument un float et qui permet de modifier le seuil de consommation d'une lampe.

public String construireRequeteFille() : c'est la redéfinition de la méthode "construireRequeteFille()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

public String constuireRequeteHistorique() : c'est la redéfinition de la méthode "constuireRequeteHistorique()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

La Classe Refrigerateur :

La Classe Refrigerateur représente le réfrigérateur de l'utilisateur, elle hérite de tout les attributs de la classe mère ObjetConnecte et contient en plus les attributs et méthodes suivantes :

```
Public class Refrigerateur extends ObjetConnecte {
private float NiveauFraicheur ;

public Refrigerateur(String nom,String type,String piece,float consoHoraire,String Droi-
tAcces, float NiveauFraicheur) ;

public Refrigerateur(Refrigerateur ref) ;

public float getNiveauFraicheur() ;
public void setNiveauFraicheur(float) ;
public String construireRequeteFille() ;
public String construireRequeteHistorique() ;
}
```

Explication des attributs

NiveauFraicheur : attribut de type float qui représente le niveau de fraîcheur d'un réfrigérateur.

Explication des méthodes

public Refrigerateur(String nom,String type,String piece,float consoHoraire,String Droi-

tAcces, float NiveauFraicheur) ;
: constructeur de la classe Refrigerateur. Il permet la création des objets de type "Refrigerateur", il prend un argument de plus que le constructeur de la classe mère ObjetConnecte qui est le niveau de fraîcheur du réfrigérateur.L'état de l'objet sera mit à "faux" donc sera "éteint" par défaut, et l'identifiant de l'objet est attribué automatiquement suivant les identifiant déjà pris.

public Refrigerateur(Refrigerateur ref) : constructeur par recopie de la classe "Refrigerateur", il a un seul paramètre "Ref" de même type de l'objet à créer qui est de type "Refrigerateur", il recopie les attributs de l'objet passé en paramètre et il les affecte à l'objet à créer.

public float getNiveauFraicheur() : méthode qui permet de récupérer le niveau de fraîcheur du réfrigérateur.

public void setNiveauFraicheur(float) : méthode qui prend un argument un float et qui permet de modifier le niveau de fraîcheur du réfrigérateur.

public String construireRequeteFille() : c'est la redéfinition de la méthode "construireRequeteFille()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

public String constuireRequeteHistorique() : c'est la redéfinition de la méthode "constuireRequeteHistorique()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

La Classe Television :
La Classe Television représente la télévision de l'utilisateur, elle hérite de tout les attributs de la classe mère ObjetConnecte et contient en plus les attributs et méthodes suivantes :

```
Public class Television extends ObjetConnecte {
private float seuilTv ;

public Television(String nom, String type,String piece,float consoHoraire, String DroitAcces,float seuilTv) ;
public Television(Television tel) ;

public float getSeuilTv() ;
public void setSeuilTv(float) ;
public String construireRequeteFille() ;
public String construireRequeteHistorique() ;
}
```

Explication des attributs

seuilTv : attribut de type float qui représente le seuil de consommation de la télévision.

Explication des méthodes

public Television(String nom, String type,String piece,float consoHoraire, String DroitAcces,float seuilTv) ; : Constructeur de la classe Television. Il permet la création des objets de type "Television", il prend un argument de plus que le constructeur de la classe mère ObjetConnecte qui est le seuil de consommation maximum de la télévision.

public Television(Television tel) : constructeur par recopie de la classe "Television", il a un seul paramètre "tel" de même type que l'objet à créer qui est de type "Television", il recopie les attributs

de l'objet passé en paramètre et il les affecte à l'objet à créer.L'état de l'objet sera mit à "faux" donc sera "éteint" par défaut, et l'identifiant de l'objet est attribué automatiquement suivant les identifiant déjà pris.

public float getSeuilTv() : méthode qui permet de récupérer le seuil de consommation de la télévision.

public void setSeuilTv(float) : méthode qui permet de modifier le seuil de consommation de la télévision.

public String construireRequeteFille() : c'est la redéfinition de la méthode "construireRequeteFille()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

public String constuireRequeteHistorique() : c'est la redéfinition de la méthode "constuireRequeteHistorique()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

La Classe Climatiseur :

La Classe Climatiseur représente le Climatiseur de la Smart Home, elle hérite de tout les attributs de la classe mère ObjetConnecte et contient en plus les attributs et méthodes suivantes :

```
Public class Climatiseur extends ObjetConnecte {
private float seuilClim ;

public Climatiseur(String nom, String type,String piece,float consoHoraire,String Droi-
tAcces,float seuilClim) ;

public Climatiseur(Climatiseur clim) ;

public float getSeuilClim() ;
public void setSeuilClim(float) ;
public String construireRequeteFille() ;
public String construireRequeteHistorique() ;
}
```

Explication des attributs

seuilClim : attribut de type float qui représente le seuil de consommation du climatiseur.

Explication des méthodes

public Climatiseur(String nom, String type,String piece,float consoHoraire,String DroitAcces,float seuilClim) : constructeur de la classe Climatiseur. Il permet la création des objets de type "Climatiseur", il prend un argument de plus que le constructeur de la classe mère ObjetConnecte qui est le seuil de consommation maximum du climatiseur.L'état de l'objet sera mit à "faux" donc sera "éteint" par défaut, et l'identifiant de l'objet est attribué automatiquement suivant les identifiant déjà pris.

public Climatiseur(Climatiseur clim) : constructeur par recopie de la classe "Climatiseur", il a un seul paramètre "clim" de même type que l'objet à créer qui est de type "Climatiseur", il recopie les attributs de l'objet passé en paramètre et il les affecte à l'objet à créer.

public float getSeuilClim() : méthode qui permet de récupérer le seuil de consommation du

climatiseur.
public void setSeuilClim(float) : méthode qui prend en argument un float et qui permet de modifier le seuil de consommation du climatiseur.

public String construireRequeteFille() : c'est la redéfinition de la méthode "construireRequeteFille()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

public String constuireRequeteHistorique() : c'est la redéfinition de la méthode "constuireRequeteHistorique()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

La Classe EnceinteConnecte :

La Classe EnceinteConnecte hérite de tout les attributs de la classe mère ObjetConnecte et contient en plus les attributs et méthodes suivantes :

```
Public Class EnceinteConnecte extends ObjetConnecte {
private float volume;
public EnceinteConnecte(String nom, String type, String piece , float consoHoraire,StringDroitAcces,float volume);

public EnceinteConnecte(EnceinteConnecte enceinte);

public float getVolume();
public void setVolume(float);
public String construireRequeteFille();
public String construireRequeteHistorique();
}
```

Explication des attributs

volume : Attribut de type float qui représente le niveau de son de l'enceinte connectée.

Explication des méthodes

public EnceinteConnecte(String nom, String type, String DroitAcces, boolean etat, float volume) : constructeur de la classe EnceinteConnecte. Il permet la création des objets de type "EnceinteConnecte", il prend un argument de plus que le constructeur de la classe mère ObjetConnecte qui est le volume de l'enceinte connectée.L'état de l'objet sera mit à "faux" donc sera "éteint" par défaut, et l'identifiant de l'objet est attribué automatiquement suivant les identifiant déjà pris.

public EnceinteConnecte(EnceinteConnecte enceinte) : constructeur par recopie de la classe "EnceinteConnecte", il a un seul paramètre "enceinte" de même type que l'objet à créer qui est de type "EnceinteConnecte", il recopie les attributs de l'objet passé en paramètre et il les affecte à l'objet à créer.

public float getVolume() : méthode qui permet de récupérer le volume de l'enceinte connectée.

public void setVolume(float) : méthode qui prend en argument un float et qui permet de modifier le volume de l'enceinte connectée.

public String construireRequeteFille() : c'est la redéfinition de la méthode "construireRequeteFille()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

public String constuireRequeteHistorique() : c'est la redéfinition de la méthode "constuireRequeteHistorique()" qui est définie comme une méthode abstraite dans la classe mère "ObjetConnecte".

La Classe CapteurPresence

La Classe CapteurPresence représente l'indicateur de localisation de l'habitant dans une pièce de la smart Home. Elle est principalement en charge de la fonctionnalité (3) du module Objet Connecté : "Capter des données intéressantes pour l'utilisateur et le système susceptibles de faire changer l'état d'un objet". Dans le cas du CapteurPresence la donnée serait donc la localisation de l'habitant dans une pièce .Elle hérite de tout les attributs de la classe mère ObjetConnecte et contient en plus les attributs et méthodes suivantes :

```
Public Class CapteurPresence extends ObjetConnecte {
private String placementUtilisateur ;
public CapteurPresence(String nom,String type, String piece,float consoHoraire,String placementUtilisateur) ;
public CapteurPresence( CapteurPresence capteur)
public String getPlacementUtilisateur() ;
public void setPlacementUtilisateur(String placement) ;
}
```

Explication des attributs

emplacementUtilisateur : Cet attribut représente la localisation de l'utilisateur dans la smart Home.

Explication des méthodes

public CapteurPresence(String nom,String type, String piece,float consoHoraire,String emplacementUtilisateur) : constructeur de la classe CapteurPresence.il permet la création des objets de type "CapteurPresence" il prend un argument de plus que le constructeur de la classe mère ObjetConnecte, qui est l'emplacement de l'utilisateur c'est à dire sa localisation. L'état de l'objet sera mit à "faux" donc sera "éteint" par défaut, et l'identifiant de l'objet est attribué automatiquement suivant les identifiant déjà pris.

public CapteurPresence(CapteurPresence capteur) : constructeur par recopie de la classe "CapteurPresence", il a un seul paramètre "capteur" de même type que l'objet à créer qui est de type "CapteurPresence", il recopie les attributs de l'objet passé en paramètre et il les affecte à l'objet à créer.

public String getPlacementUtilisateur() : méthode qui permet de récupérer la localisation de l'habitant dans sa Smart Home.

public void setPlacementUtilisateur(String placement) : méthode qui prend en argument un String et qui permet de modifier la localisation de l'habitant dans sa Smart Home.

3.1.3 La classe Routeur

Cette classe est une classe qui assure la communication entre les objets (donc une communication inter-objets), et la communication entre les objets et les autres modules (Calcul et Optimisation/Objets Connectés, Gestionnaire de données/Objets Connectés).
Elle se particularise par le fait d'être une classe Singletone :
Le Singleton est un patron de conception de création qui s'assure de l'existence d'une seule instance de

son genre et fournit un unique point d'accès vers cet objet.Elle sera chargée directement au lancement de l'application.
Les autres classes et modules du logiciel pourront donc appeler cette même instance avec la méthode *Routeur getInstance()* afin de pouvoir utiliser les différentes fonctions du Routeur.
Elle est en charge de la fonctionnalité (1) du module Objet Connecté citée plus haut.

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public class Routeur {
private static Routeur instance =new Routeur();
private Map<Integer,ObjetConnecte> lesObjets;

private Routeur();

public static Routeur getInstance();

public void ajouterRefObjet(ObjetConnecte obj);
public void supprimerRefObjet(int IdObjet);
public void envoyerAlerte(String alerte, int IdObj);
public void envoyerReglage(Reglage reglage, int IdObj);
public void envoyerRequete(String requete);
public ObjetConnecte fournirReference(int IdObj);
public ArrayList<ObjetConnecte> fournirReference(int IdObj[]);
}
```

Explication des attributs

instance : attribut de type static Routeur, qui est une particularité du "Design Pattern Singleton". Il représente une instance unique de la classe Routeur. L'instanciation de l'attribut avec le constructeur *Routeur()* dès sa déclaration assure que notre instance Singletone existe dès le chargement des classes JAVA au lancement de l'application. Cela garanti que même un appel double au getInstance() (au même moment dans un environnement multi-thread) ne créera pas deux instances de cette classe.

lesObjets : une map qui contient toutes les références des objets connectés associées à un identifiant entier. Le choix de la structure de données Map contrairement à une structure ArrayList par exemple repose sur un argument en particulier : Il est vrai qu'une ArrayList avec une bonne récupération de données (objets connectés ordonnés suivant l'identifiant) est une bonne solution en général, mais l'utilisateur pourra aussi supprimer des Objets connectés à travers l'application, de ce fait, en prenant une structure ArrayList, nous ne pourrons plus accéder à nos objets avec une complexité O(1),(les identifiants des autres objets ne peuvent pas être modifiés). De ce fait le choix d'une Map avec comme clé l'identifiant de l'objet récupéré avec la méthode *getIdentifiant* est donc le meilleur choix.

Explication des méthodes

Routeur() : Constructeur sans paramètres de la classe Routeur qui sera appelé une seule fois au chargement des classes au moment du lancement de l'application. Il instanciera l'attribut "lesObjets" de type Map en une Hashmap, avec comme clé l'identifiant de l'objet de type entier et en valeur l'objet connecté de type *ObjetConnecte*. Son encapsulation privée permet d'interdire l'accès au constructeur (nouvelle instanciation).

public static Routeur getInstance() : permet de récupérer une instance de la classe Routeur à partir de nos objets connecté et à partir des autres modules du logiciel. Elle retourne l'instance unique de la classe routeur contenu dans l'attribut instance.

void ajouterRefObjet (ObjetConnecte obj) : méthode qui permet d'ajouter une référence sur un objet connecté dans la Map lesObjets. Elle sera appelée lors de l'intégration de nouveaux objets ou la récupération des objets connectés de l'utilisateur lors du lancement de l'application.

public void supprimerRef(int idObjet) : méthode qui permet de supprimer une référence sur un objet connecté dans la Map lesObjets. Elle sera appelée lors de la suppression d'un objet par l'utilisateur.

envoyerAlerte(String alerte, int idObjet) : méthode qui permet d'acheminer l'alerte qui provient de l'objet connecté à partir de la fonction ObjetConnecte.envoyerAlerte() vers le module calcul et optimisation en appelant la fonction optimiserConsommationUnique(Objet obj).

public void envoyerReglage(Reglage reglage) : méthode qui permet d'envoyer un nouveau Réglage sur un objet connecté.Elle prend en paramètre un objet de type Reglage (classe définie dans le module calcul et optimisations). La fonction appelle ensuite la méthode *appliquerReglage(Reglage R)* présente dans la classe ObjetConnecte sur l'objet concerné (dont on a spécifié l'identifiant).

public void envoyerRequete(String requete) : Cette méthode est appelé par un objet connecté lors de la modifcation de l'un de ses attributs, elle permet d'appeler une instance de *BDDSingleton* et d'envoyer la requête à travers la méthode decisionRequete(String requete) de la classe BDDSingleton.

public ObjetConnecte fournirReference(int IdObj) : Cette méthode permet de retourner une référence de l'objet connecté ciblé avec l'identifiant donné en paramètres.

public ArrayList<ObjetConnecte> fournirReference(int[] idObj) : Cette méthode permet de retourner des références sur les objets connectés ciblés avec les identifiants donné en paramètres grâce au tableau d'identifiants.

3.2 Module Calcul et Optimisation

Ce module permet d'effectuer tous les calculs de consommation dont l'utilisateur a besoin et effectue les réglages d'optimisations.Il est représenté par le diagramme de classes suivant :

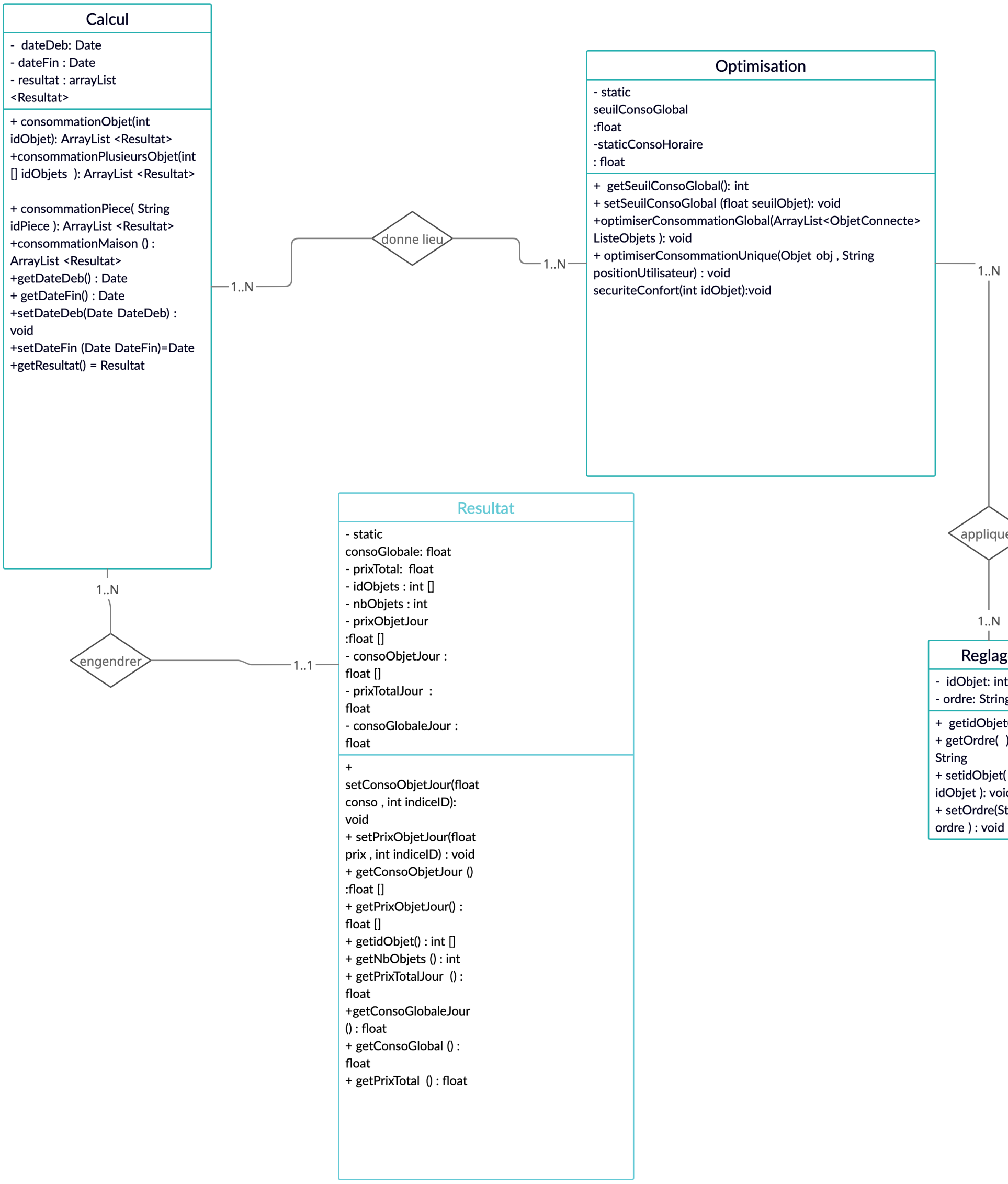


FIGURE 3 – Diagramme de Classes du Module Calcul et Optimisation

Explication des cardinalités :

Cardinalité entre Calcul et Optimisation : Un calcul peut générer plusieurs optimisations et une optimisation peut être générer par plusieurs calculs.

Cardinalité entre Reglage et Optimisation : Une optimisation peut générer plusieurs réglages et chaque réglage peut être généré par plusieurs optimisations.

Cardinalité entre Calcul et Resultat : Un calcul peut générer plusieurs résultat et un résultat est généré par un seul calcul.

3.2.1 La classe Resultat

Cette classe est une classe qui permet de stocker les détails de résultat d'un calcul de consommation d'un objet ou plusieurs par jour et aussi sur toute la période concernée. (la consommation énergétique et le prix équivalent de chaque objet par jour, et la consommation globale et le prix total de l'ensemble des objets concernés par rapport à la période concernée).

```
Public class Resultat {

private int [] idObjets;
private int nbObjets;
private float[ ] prixObjetJour;
private float[ ] consoObjetJour;
private float prixTotalJour;
private float consoGlobaleJour;
private static float consoGlobale;
private static float prixTotal;

public Resultat(int[ ] idObjets,int nbObjets);

public void setConsoObjetJour(float conso,int indiceID);
public void setPrixObjetJour(float prix,int indiceID);
public float[ ] getConsoObjetJour();
public float[ ] getPrixObjetJour();
public int[ ] getIdObjets();
public int getNbObjets();
public float getPrixTotalJour();
public float getConsoGlobaleJour();
public float getConsoGlobale();
public float getPrixTotal();
}
```

Explication des attributs

private int[ ] idObjet : Attribut de type int[ ], qui représente un tableau d'identifiants des objets concernés.

private int nbObjets : Attribut de type int qui représente le nombre d'objets concernés par le calcul.

private float[ ] prixObjetJour : Attribut de type "float[ ]" qui représente un tableau de float

chaque élément du tableau "prixJour[i]" représente le prix équivalent à la consommation de l'objet "idObjets[i]" par jour, sa taille est "nbObjets".

private float[ ] consoObjetJour : Attribut de type "float[ ]" qui représente un tableau de nombres réel chaque élément du tableau "consoJour[i]" représente la consommation énergétique de l'objet "idObjet[i]" par jour, sa taille est "nbJours".

private float prixTotalJour : Attribut de type float, qui représente le prix Total d'un ou plusieurs objets du jour concerné.
private float consoGlobaleJour : Attribut de type float, qui représente la consommation d'un ou plusieurs objets du jour concerné.

private int[ ] idObjet : Attribut de type float, qui représente la consommation globale sur la période demandée par l'utilisateur.

private float prixTotal : Attribut de type float, qui représente le prix total équivalent à la consommation globale sur la période demandée par l'utilisateur.

Explication des méthodes

public Resultat(int[ ] idObjets,int nbObjets) : Constructeur de la classe Resulatat, il permet de créer et d'instancier des objets de type "Resultat".

public void setConsoObjetJour(float conso,int indiceID) : méthode qui permet de modifier la consommation du jour de l'objet qui est a comme indice "indiceID" dans le tableau "idObjets" " avec la valeur "conso" (consoObjetJour[indiceID] = conso), et modifier la valeur de l'attribut "consoGlobale" (consoGlobale = consoGlobale + conso).

public void setPrixObjetJour(float prix,int indiceID) : méthode qui permet de modifier le prix équivalent à la consommation du jour de l'objet qui est à l'indice "indiceID" dans le tableau "idObjets" " avec la valeur "prix" (prixObjetJour[indiceID] = prix ;)., et modifier la valeur de l'attribut "prixGlobal" (prixTotal = prixTotal + prix).

public float[ ] getConsoObjetJour() : méthode qui permet de récupérer le tableau de consommation d'un seul jour des objets concernés par le calcul.

public float[ ] getPrixObjetJour() : méthode qui permet de récupérer le tableau des prix équivalents à la consommation d'un seul jour des objets concernés par le calcul.

public int[ ] getIdObjet() : méthode qui permet de récupérer le tableau des identifiants des objets concernés par le calcul.

public int getNbJours() : méthode qui permet de récupérer le nombre d'objet concernés par le calcul.

public float getConsoGlobaleJour() : méthode qui permet de récupérer la consommation des objets concernés par le calcul sur une seule journée.

public float getPrixTotalJour() : méthode qui permet de récupérer le prix équivalent à la consommation globale des objets concernés par le calcul sur une seule journée.

public float getConsoGlobale() : méthode qui permet de récupérer la consommation des objet concernés par le calcul sur toute la période demandée.

public float getPrixTotal() : méthode qui permet de récupérer le prix équivalent à la consommation globale des objet concernés par le calcul sur toute la période demandée.

3.2.2 La classe Calcul

Cette classe s'occupe des calculs de consommation d'énergie de notre application, soit des calculs demandé par l'utilisateur pour pouvoir suivre sa consommation par jour ou bien par le système dans le but de réaliser des optimisations .

```
import java.util.ArrayList;

Public class Calcul {

private Date dateDeb;
private Date dateFin;
private ArrayList<Resultat> resultat;

public Calcul();

public ArrayList<Resultat> consommationObjet (int idObjet);
public ArrayList<Resultat> consommationPlusieursObjets(int[ ]idObjets);
public ArrayList<Resultat> consommationPiece (String idPiece);
public ArrayList<Resultat> consommationMaison ();
public Date getDateDeb();
public Date getDateFin();
public void setDateDeb(Date DateDeb);
public void setDateFin (Date DateFin);
public Resultat getResultat();
}
```

Explication des attributs

dateDeb : Attribut de type Date qui représente la date de début de consommation à partir de laquelle l'utilisateur veut calculer la consommation d'un objet connecté ou plusieurs ,d'une pièce ou de la maison.

dateFin : Attribut de type Date qui représente la date de fin de consommation d'un objet à partir de laquelle l'utilisateur va calculer la consommation énergétique d'un objet connecté ou plusieurs ,d'une pièce ou de la maison.
A partir de ces deux attributs, l'utilisateur peut calculer la consommation énergétique d'un ou de plusieurs objets lors d'une période qui va de dateDeb à dateFin.

resultat : Attribut de type ArrayList<Resultat> qui représente une liste des "Resultat", et chaque élément de la liste représente un objet "Resultat" qui permet de stocker les détails de résultat d'un calcul de consommation par jour d'un objet ou plusieurs (consommation énergétique du jour et son prix équivalent, consommation énergétique globale du jour et son prix équivalent, consommation énergétique globale et son prix équivalent sur toute la période), on a choisi de créer une classe pour le

résultat dans le but d'avoir beaucoup d'informations sur le résultat d'un calcul.

Explication des méthodes

public Calcul() : Constructeur par défaut de la classe Calcul. Il permet la création d'objets de type Calcul et la création de la liste "resultat".

public ArrayList<Resultat> consommationObjet(int idObjet) :méthode qui calcule la consommation d'un unique objet "idObjet" passé en paramètres.Cette méthode va appeler la méthode requeteSelection(String requete) de la classe " BDDSingleton", celle-ci applique une requete de selection "requete" sur notre base de données, cette dernière méthode va retourner un "ResultSet" qui va contenir la réponse de la requête "requete" qui représente les données nécessaires pour faire le calcul (consoHoraire, prix coresspondant à 1 Watt, durée d'allumage de l'objet par rapport à "DateDeb" et "DateFin") Elle renvoie en suite la liste de nos resultat journaliers.Elle effectue donc la fonctionnalité (1) du module calcul et optimisation :"Calculer la consommation d'un ou plusieurs objets en se basant sur les données récoltées".

public ArrayList<Resultat> consommationPlusieursObjets(int[ ] idObjets) : méthode qui calcule la consommation d'un ensemble d'objets passés en paramètres. Elle utilisera la méthode consommationObjet(int idObjet) sur chaque élément du tableau "idObjets".Elle renvoie à la fin la liste des résultats journaliers.Elle effectue donc la fonctionnalité (1) : Calculer la consommation d'un ou plusieurs objets en se basant sur les données récoltées".

public ArrayList<Resultat> consommationPiece(String idPiece) méthode qui calcule la consommation énergétique totale d'une pièce de la maison,cette fonction va utiliser la méthode consommationPlusieursObjets(int [ ] idObjets) tel que " idObjets" représente la liste d'objets qui composent la pièce concernée.

public ArrayList<Resultat> consommationMaison() : méthode qui calcule la consommation totale de la SmartHome.Elle utilisera la méthode consommationPlusieursObjets(int[ ] idObjets) avec comme paramètre tous les objets composants la maison.

public DATE getDateDeb() : méthode qui permet de récupérer l'attribut dateDeb.

public Date getDateFin() : méthode qui permet de récupérer l'attribut dateFin.

public void setDateDeb(DATE dateDeb) :méthode qui permet de modifier l'attribut dateDeb.

public Date setDateFin() : méthode qui permet de modifier l'attribut dateFin.

public ArrayList<Resultat> getResultat() : méthode qui permet de récupérer la liste "resultat", qui va contenir les information du résultat de calcul de consommation demandé.

3.2.3 La classe Optimisation

Cette classe est une classe qui effectue l'optimisation d'énergie sur un objet ou plusieurs, les méthodes de cette classe se déclenchent à partir du Routeur : il reçoit une alerte depuis objet connecté ou bien calcul, il va déclencher les méthodes de cette classe en utilisant des nouveaux réglage sur les objets dans le but de les optimiser.

```
Public class Optimisation {

private static float seuilConsoGlobal ;
private static float ConsoHoraireGlobale ;
public Optimisation() ;



public void optimiserConsommationGlobale(String positionUtilisateur) ;
public void optimiserConsommationUnique(int idObjet,String positionUtilisateur) ;
public void securiteConfort(int idObjet) ;
public void incrementer(float consoHoraireObjet) ;
public void décrementer(float consoHoraireObjet) ;
public void setSeuilConsoGlobale(float seuilConsoGlobale) ;
public static float getSeuilConsoGlobale() ;
public static float getConsoHoraire() ;

}
```

Explication des attributs

attributs : explication.

private static float seuilConsoGlobal : Attribut de type "float", il représente le seuil de consommation de la maison à ne pas dépasser, on récupère la valeur de cet attribut de gestionnaire de données en utilisant la méthode "requeteSelection(requete)" de la classe "BDDsingleton" sur la table "Configuration". Si cet attribut à été dépasser, la fonction "optimiserConsoGlobale()" de cette classe va se déclencher pour optimiser la consommation de la maison.

private static float ConsoHoraireGlobale ; Attribut de type "float", il représente la consommation de la maison par heure, on modifie la valeur de cet attribut grâce aux deux méthodes de cette classe "decrementer(float consoHoraireObjet)" et "incrementer(float consoHoraireObjet)", la valeur de cette attribut est la somme des consommations horaires des objets qui sont allumées.Si la valeur de cet attribut est dépassé (autrement dit si le circuit énergétique est trop sollicité par trop d'appareils allumés en même temps) on est obligé d'éteindre les objets qu'on a le droit d'éteindre.

Explication des méthodes

public Optimisation() : Constructeur de la classe Optimisation, permet la création d'objet de type Optimisation dans le but d'utiliser les fonction de cette classe qui optimisent la consommation d'un objet ou plusieurs car son/leurs seuil(s) de consommation a(ont) été dépassé.

public void optimiserConsommationGlobale(String positionUtilisateur) Méthode qui se déclenche quand le seuil de consommation énergétique de la maison fixé auparavant par l'utilisateur ou bien l'attribut "seuilConsoGlobale" est dépassé, elle permet de récupérer la liste des références des objets de notre SmartHome depuis le routeur,puis elle va appliquer des nouveaux réglages sur chaque objet de la liste selon la position de l'utilisateur qu'on va récupérer depuis le routeur, grâce à la création des objets de la classe "Reglage" qui va nous permettre d'avoir des nouveaux réglages sur les objet et les appliquer sur ces derniers grâce à la méthode "appliquerReglage(Reglage r)" de la classe "Objet-Connecte" dans le but d'optimiser la consommation de chaque objet connecté puis la consommation globale de la maison. Elle fait donc la fonctionnalité(2) du module calcul et optimisation : " Envoyer l'ordre pour changement d'état d'un ou plusieurs objets".

public void optimiserConsommationUnique(int idObjet,String positionUtilisateur) : Méthode qui se déclenche après la réception d'un signal de la part du Routeur l'informant que le seuil de consommation d'un objet connecté ou la durée de son allumage par défaut a été dépassé.Elle prend en argument l'identifiant de l'objet à optimiser et la position de l'utilisateur(accord ou pas de l'utilisateur) qu'on va . Elle permet de modifier le réglage de l'objet concerné, en créant un objet de la classe "Reglage" et en l'appliquant sur l'objet avec sa référence qu'on va récupérer depuis le routeur grâce à son identifiant, grâce à la méthode "appliquerReglage(Reglage r)" dans le but d'optimiser sa consommation.Elle représente donc la fonctionnalité(2) du module calcul et optimisation : " Envoyer l'ordre pour changement d'état d'un ou plusieurs objets".

public void securiteConfort(int idObjet) : Méthode qui applique la solution d'optimisation sur l'objet.Elle effectue donc la fonctionnalité (5) du module calcul et optimisation :"Appliquer directement la solution trouvée en envoyant de nouveaux réglages aux objets connectés".

public void setSeuilConsoGlobale(float seuilConsoGlobale) : Elle permet de modifier la valeur de l'attribut "seuilConsoGlobal".

public void incrementer(float consoHoraireObjet) : Elle permet de rajouter au consoHoraire-Globale la valeur de consoHoraireObjet. Elle est déclenché lorsqu'on allume un objet connecté

public void decrementrer(float consoHoraireObjet) : Elle permet de soustraire la consommation horaire de l'objet à l'attribut consoHoraireGlobale . Elle se déclenche lorsqu'on éteint un objet.

public void getSeuilConsoGlobale() : Elle permet de récupérer l'attribut SeuilConsoGlobale.

public void getConsoHoraireGlobale() : Elle permet de récupérer l'attribut consoHoraireGlobale.

3.2.4 La classe Reglage

Cette classe est une classe qui permet d'effectuer un réglage sur un objet.Les méthodes de cette classe seront utilisés par la classe Optimisation dans le but d'optimiser.

```
Public class Reglage {
private int idObjet ;
private String ordre ;

public Reglage(int id,String ordre) ;

public int getIdObjet() ;
public String getOrdre() ;
public void setIdObjet(int IdObjet) ;
public void setOrdre(String Ordre) ;
}
```

Explication des attributs

private idObjet : Attribut de type int qui définit l'identifiant de l'objet sur lequel on applique un réglage.

private ordre : Attribut de type String qui définit l'ordre de réglage à appliquer.

Explication des méthodes

public int getIdObjet() : méthode qui permet de récupérer l'identifiant de l'objet auquel s'applique le réglage.

public String getOrdre() : méthode qui permet de récupérer l'ordre de réglage à appliquer sur l'objet.

public void setIdObjet(int IdObjet) : permet de modifier l'identifiant de l'objet auquel on applique le réglage.

public void setOrdre(String Ordre) : permet de modifier l'ordre de réglage à appliquer sur l'objet.

3.3 Module Gestionnaire de données

Ce module permet de sauvegarder toutes les données de nos objets connectés.

3.3.1 Choix de l'implémentation de la base de données

Étant donné le fait que notre projet doit disposer d'une base de données extensible et qui peut gérer plusieurs utilisateurs avec des configurations propres, et étant donné le nombre important d'historiques que l'on doit définir dans notre base de données, et la multitude d'insertions qui en découle (afin d'avoir une simulation proche de la réalité), notre choix s'est donc tourné vers une implémentation à base d'un SGBD.
En effet, cette approche nous a permis de définir un modèle entité/association qui répond aux besoins de notre projet, et qui nous permettra une fois les tables créées d'accéder d'une manière structuré et simple aux données recherchés afin d'opérer une modification ou une sélection de données.
De plus, les SGBD proposent des mécanismes de contraintes d'intégrité, qui nous permettent de garantir l'authenticité de notre donnée, et son unicité.
Enfin, cette approche nous donne la possibilité d'implémenter et de remplir une base de données unique regroupant les données de tous nos utilisateurs.

3.3.2 Choix du SGBD

À l'écriture de notre cahier des charges nous avons énoncé certaines contraintes et perspectives d'améliorations de notre application. À savoir :
- L'utilisateur doit pouvoir accéder facilement à notre application.
- Notre application doit être portable et éligible à une futur intégration sur mobile.

- Notre application doit pouvoir assurer la sauvegarde et la récupération des données de l'utilisateur.
- Notre gestionnaire de données doit fournir de manière simple les données nécessaires aux autres modules du logiciel, à savoir le module **Calcul et Optimisation** et le module **Interface Graphique**.

Notre choix c'est donc tourné vers **SQLite** *Réf(1)*. En effet, contrairement aux serveurs de bases de données traditionnels, comme MySQL ou PostgreSQL, sa particularité est de ne pas reproduire le schéma habituel client-serveur mais d'être directement intégrée aux programmes. L'intégralité de la base de données (déclarations, tables, index et données) est stockée dans un fichier indépendant de la plateforme. *Réf(3)*
De plus, SQLite est idéal pour les petites applications, il est conçu pour être performant même dans des environnements à faible ressource (utilisation sur mobile par exemple), il n'a pas besoin d'un serveur pour fonctionner : nous n'aurons donc pas besoin d'avoir SQL sur notre machine pour exécuter l'application ce qui est un des avantages majeurs du SGBD. *Réf(2)*
Enfin, il existe différents outils de conception graphiques pour faciliter la visualisation des tables et la création du format adéquat (fichier au format ".db" à mettre à coté du projet java) dont le logiciel SQLite studio.
Dans notre application nous allons utiliser la version 3.30.1 de SQLite, qui est la version la plus récente de SQLite. Nous allons donc fournir lors de la remise de notre application le fichier **sqlite-jdbc-3.30.1.jar** nécessaire au fonctionnement de notre application.

3.3.3 Modèle Entité/Association

L'ensemble des classes que nous présenterons ci-dessous appartiennent au module(package) Gestionnaire de données.

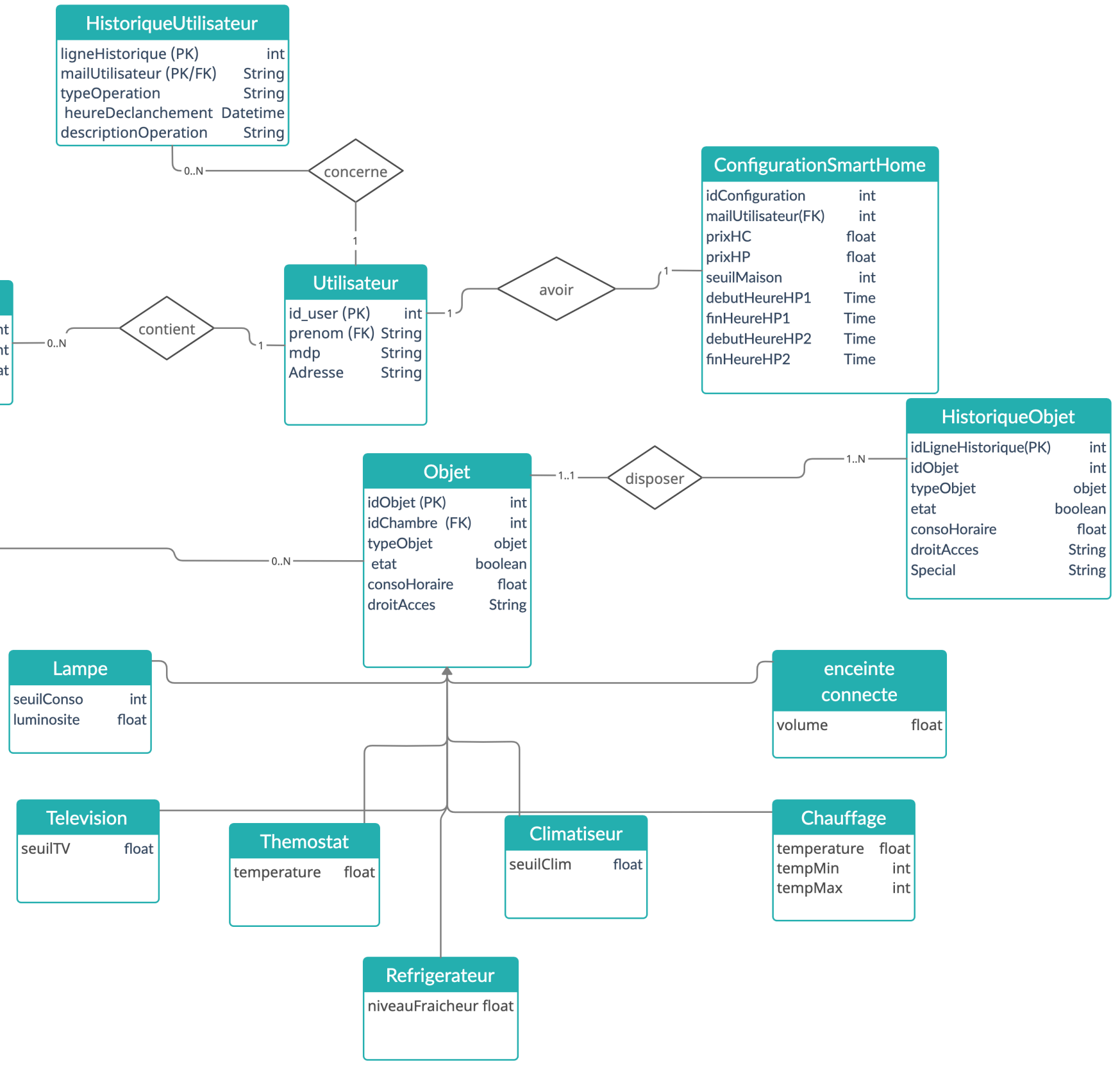


FIGURE 4 – diagramme$_d e_b dd$

3.3.4 La classe Connexion

Cette classe est une classe qui permet de lier le programme JAVA à SQLite.

```
import java.sql.*;

Public class Connexion {
private final String BDPath = "BD_SGE.db";
private Connection connection;
private Statement statement;

public Connexion(){
connect();
};

public void connect();
public ResultSet query(String requete);
public void close();
}
```

Explication des attributs

BDPath : nom du fichier dans lequel sera stocker la base de données.

connection : représente l'attribut qui permettra de connecter le *DriverManager* SQL à notre programme.

statement : représente l'attribut qui permettra d'executer nos requêtes. Il sera initialiser a partir de l'attribut *connection*. une fois le le constructeur appelé et l'attribut *connection* instancié, on retournera le *statement* correspondant dans notre attribut (avec l'utilisation de la fonction prédéfinie *connection.createStatement()*), et qui nous permettra d'exécuter nos requêtes sur la base de données une fois la connexion établie (avec la fonction prédéfinie *statement.executeQuery(String requete)*).

Explication des méthodes

public Connexion() : constructeur de la classe Connexion. Dans notre classe le constructeur appelle directement la méthode *connect()*. Nous avons pris ce choix afin de garantir la connexion avec notre base de données dés l'instanciation de la classe Connexion (au moment du lancement de l'application).

void connect() : ouvre la connexion entre notre application et la base de données SQLite. Elle permet l'instanciation des attributs *connection* et *statement*.

ResultSet query(String) : effectue la requête donnée en paramètre dans notre base de données et retourne un ResultSet qui est un type définie dans la librairie *java.sql.ResultSet* et qui représente le résultat d'une requête SQL.

void close() : ferme la connexion entre JAVA et la base de données SQLite.

3.3.5 La classe BDDSingleton

Cette classe garantie la communication entre nos différents modules et notre base de données SQL (réception de requêtes et envoie de résultats). Elle permet de gérer les différentes requêtes émises et de traiter leurs réponses et les exceptions générées dans le cas échéant.
- Le patron de conception "Singleton" est présent dans notre classe car l'accès à notre base de données doit être unique et centralisé pour tout les modules de notre logiciel.

```
Public class BDDSingleton {
private static BDDSingleton instance = new BDDSingleton() ;
private Connexion connexion ;
private float prixHC ;
private float prixHP ;
private String mail_user ;

private BDDSingleton() ;

public static BDDSingleton() ;
public void fermerConnexion() ;
public void decisionRequete(String requete) ;
public ResultSet requeteSelection(String requete) ;
public void requeteInsertion(String requete) ;
public void requeteModification(String requete) ;
public void requeteSuppression(String requete) ;
public void donneeUtilisateur(Donnee data) ;
public void setPrixHC_PrixHP() ;
public void sauvegarderTout() ;
public boolean verifierIdentifiants(String mailUtilisateur, String mdp) ;
}
```

Explication des attributs

instance : instance de la classe BDDSingleton qui permet de la récupérer pour la modifier dans une autre classe.
Les autres classes et modules du logiciels pourront donc appeler cette même instance avec la méthode *BDDSingleton getInstance()* afin de pouvoir appliquer les différentes requêtes sur notre base de données. L'instanciation de l'attribut dès sa déclaration garantie la présence du singleton dès le chargement des classes java au moment de l'exécution de l'application.

connexion : représente un objet de type Connexion (la classe présentée plus haut) servira à la mise au point des drivers SQL et la connexion entre le programme et la base de données. Et à l'application finale des requêtes avec la fonction *ResultSet query(String)*.

prixHC : détermine le prix du Watt/heure lors des périodes creuses.Cet attribut est à récupéré directement à partir de la table ConfigurationSmartHome une fois l'identifiant de l'utilisateur connu.

prixHP : détermine le prix du Watt/heure lors des périodes pleines.Cet attribut est à récupéré directement à partir de la table ConfigurationSmartHome une fois l'identifiant de l'utilisateur connu.

mail_user : représente l'identifiant de l'utilisateur connecté à l'application.On le récupérera au moment de la vérification des identifiants avec la méthode *verifierIdentifiants(String mailUtilisateur, String mdp)*.

Explication des méthodes

private BDDSingleton() : représente le constructeur de la classe BDDSingleton, il sera appelé de manière unique pour l'instanciation de l'attribut *instance* au moment de l'exécution.

static BDDSingleton getInstance() : représente la méthode qui nous permettra d'invoquer l'instance unique de notre classe BDDSingleton, dans les autres packages du programme pour pouvoir

appliquer des méthodes sur notre classe.

void fermerConnexion() : ferme la liaison avec SQLite, elle sera déclenchée lors de la fermeture de l'application.

public void decisionRequete(String requete) : représente la méthode permettant de différencier le type de requête qui est envoyé à la classe BDDSingleton, elle appellera en analysant le mot clé correspondant de la requête, exemple (select, insert, set,delete) la méthode correspondante.

ResultSet requeteSelection(String) : retourne le résultat de la requête entrée en paramètre. La méthode aura pour rôle d'executer la méthode *query(String requete)* sur l'attribut connexion de la classe.

void requeteInsertion(String requete) : exécute la requête donnée en paramètre dans la base de données SQLite. Elle est en charge de la fonctionnalité numéro (2) du module gestionnaire de donnée cité plus haut :"Sauvegarder et mettre à jour les données de métmation/états des objets".

void requeteModification(String requete) : exécute la requête donnée en paramètre dans la base de données SQLite.Elle est en charge de la fonctionnalité numéro (2) du module gestionnaire de donnée cité plus haut :"Sauvegarder et mettre à jour les données de consommation/états des objets".

void requeteSuppression(String requete) : exécute la requête donnée en paramètre dans la base de données SQLite. La méthode ne retourne rien. La méthode ne retourne rien. Cette classe prends en charges une partie de la fonctionnalité (1) du module Gestionnaire de Données.

public void donneeUtilisateur(Donnee data) : représente la méthode permettant la réception de la donnée de l'utilisateur, c-à-d toutes les manipulations qu'il a demandé au niveau de l'interface, que ce soit des calculs, des historiques, des ajouts, modification d'objets. La méthode stock ensuite la ligne d'historique correspondante dans la table HistoriqueUtilisateur après avoir généré la requête grâce à la méthode *construireRequete()* de la classe BDDSingleton.

void setPrixHC_PrixHP(int) : définie les horaires creuses et les horaires pleines.La méthode sera appelée après la vérification des identifiants.

void sauvegarderTout() : la méthode se déclanche quand l'utilisateur appuie sur le bouton sauvegarder, elle invoque l'instance du Routeur, qui va fournir les références des objets et puis elle va sauvegarder les données pour chaque objet en insérant dans l'historique de ces derniers. Dans le cas contraire, la méthode retourne "Faux".

boolean verifierIdentifiants(String mailUtilisateur,String mdp) vérifie le nom et le mot de passe de l'utilisateur lors sa connexion. Elle modifiera la valeur de l'attribut mailUtilisateur dans le cas d'une vérification validée et retournera "Vrai" et appellera la méthode *setPrixHC_PrixHP()*.

3.3.6 La classe Donnee

Cette classe est une classe qui permet de stocker les historiques de commandes de l'utilisateur dans le cas d'une consultation d'historique, d'un calcul de consommation, d'un ajout d'objet ou de pièces, d'une modification d'attributs, d'une suppression d'objets.

```
Public class Donnee {
private String typeOperation ;
private LocalDateTime dateHeureDeclanchement ;
private String descriptionOperation ;

public Donnee() ;
public Donnee(String typeOperation, LocalDateTime dateHeure, String description) ;



public void setTypeOperation(String type) ;
public void setdateHeureDeclanchement(LocalDateTime dateHeure) ;
public void setDescriptionOperation(String Description) ;

public String getTypeOperation() ;
public LocalDateTime getDateHeureDeclanchement() ;
public String getDescriptionOperation() ;

public String construireRequete() ;
public String toString() ;
}
```

Explication des attributs

typeOperation : détermine le type de l'opération appliqué par l'utilisateur (calcul,ajout,modification,suppressio

dateHeureDeclanchement : détermine la date et l'heure à laquelle l'utilisateur a déclenché l'opération.

descriptionOperation : comporte une description de l'opération et de son résultat final dans le cas d'un calcul par exemple.

Explication des méthodes

Public Donnee() : représente le constructeur sans paramètres de la classe Donnee.

Public Donnee(String typeOperation, DateFormat dateHeure, String description) : représente le constructeur paramétré de la classe Donnee.

public void setTypeOperation(String type) : setter de l'attribut typeOperation.

public void setDateHeureDeclanchement(LocalDateTime dateHeure) : setter pour modifier l'attribut "dateHeureDeclanchement".

public void setDescriptionOperation(String description) : setter pour modifier la description.

public String getType() : retourne l'attribut typeOperation.

public LocalDateTime getDateHeureDeclanchement() : retourne l'attribut dateHeureDeclanchement.

public String descriptionOperation() : retourne l'attribut "descriptionOperation".

public String construireRequete() : méthode qui permet de construire la requête correspondante à l'insertion de la donnée dans la table "HistoriqueUtilisateur" de la base de donnée SQL.

3.4 Module Interface graphique

3.4.1 Choix de la bibliothèque graphique

Après avoir consulté différentes bibliothèques graphiques utilisant principalement le langage JAVA notamment LWJGL/Slick2D,Swing,et JAVAFX,nous avons choisi d'utiliser la bibliothèque **JAVAFX**.On trouve que c'est la plus adaptée pour la réalisation de l'interface utilisateur de notre projet pour différentes raisons :

- Elle a un panel de choix de librairies externes .

- Elle est portable, c'est à dire qu'on pourra l'exécuter sur différents systèmes d'exploitations et supporte aussi l'intégration mobile et web.

- Elle propose un large choix de types de graphes qui nous seront utiles lors de la réalisation des graphes d'évolution de la consommation énergétique de la Smart Home.

- Notre interface doit être interactive et doit proposer une large palette de choix à l'utilisateur pour gérer son environnement SmartHome, l'utilisation du ***Scene Builder*** nous sera d'une grande utilité afin de remplir les différentes exigences d'une manière simple et efficace.

Mise en Oeuvre

Important : Toutes les méthodes qui prennent un évenement en paramètre "Event Listner" ne seront pas mentionnés dans les classes ci-dessous, car l'implémentation de ces méthodes est prédéfinis dans javaFX, par conséquent toutes les méthodes qui seront abordés sont des méthodes que nous avons personnellement défini.
le model view controller (MVC) intégré et utilisé par javafx permet une implémentation sans constructeur, en effet, nos controlleurs représentent des vues de nos fichiers FXML. Chaque controlleur, aura son fichier FXML associé, au lancement de l'application les fichiers FXML sont chargés.
Extrait du code fxml qui fait l'implémentation d'un 'Event Listner' d'un Button :
<Button mnemonicParsing="false" onMouseClicked="AjouetObjet" text="Ajouter Objet">

L'utilisation de JAVAFX et du Scene Builder dans notre application inclut l'utilisation des fichiers FXML qui définisssent le squelette de nos scènes,des fichiers CSS qui permettent l'intégration de figures et de modifications de styles ,et des classes Controlleurs qui nous permettent de gérer les interactions avec l'utilisateur et de déclencher les méthodes appropriées.

1-Fichier FXML : FXML est un format de données textuelles, dérivé du format XML et qui décrit l'interface graphique qui sera ensuite chargé dynamiquement dans l'application.

2-Contrôleur : Un contrôleur est une classe externe au fichier FXML, généralement écrite en Java, qui est chargée par le chargeur FXML en même temps que le fichier FXML. Cette classe permet de manipuler les entités nommées décrites dans le fichier FXML, d'y accéder depuis le reste de l'application ou même de modifier le contenu de l'arborescence décrite dans le FXML.

3-fichier CSS : Le CSS correspond à un langage informatique permettant de mettre en forme des pages web/Logiciel (HTML ou XML)

3.4.2 La classe Main

Cette classe contient la méthode principale 'main' et la méthode 'Start' qui va permettre le chargement de la fenêtre d'identification. Et permettre à notre utilisateur de se connecter.

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.fxml.FXMLLoader;

public class Main extends Application{
 public void Start(Stage primaryStage);
 public static void main(String[] args) {
      launch(args);
}
}
```

Explication des méthodes

public void start(Stage primaryStage) : Méthode qui prend un argument de type 'Stage' et qui permet de charger la fenêtre d'identification.
public void main(String[] args) : Méthode qui permet de lancer notre application avec des arguments passés en ligne de commande , elle utilise la méthode 'launch' qui permet de lancer la méthode start().

3.4.3 La classe GestionObjetsControlleur

Cette classe permet de gérer les objets :ajout,réglage et visualisation.Elle permet de créer une pièce, un objet, modifier le réglage des objets déjà créés,la visualisation des pièces ainsi que les objets de cette pièce et leurs réglages.Elle est en charge de la fonctionnalité (1) du Module Interface Graphique

"Afficher la liste d'objets connectés qui constituent la Smart Home sur l'interface".

```
import javafx.scene.layout.StackPane;

public class GestionObjetsControleur {
private StackPane StackObjets;
private StackPane StackObjetsConfiguration;
private StackPane StackPieces;

public void actionSurObjet(int idObjet);
public void reglerObjet(int idObjet, int valeurModifiee);
public void ajouterObjet();
public void ajouterPiece ();
public void construire();
}
```

Explication des attributs

StackObjets : Attribut de type StackPane qui va contenir toutes les représentations graphiques des objets d'une pièce sélectionnée.
La structure de donnée est incluse dans le package *javafx.scene.layout*, elle permet l'ajout de différentes composantes d'interface de manière à en visualiser qu'une sous partie.

StackObjetsConfiguration : Attribut de type StackPane qui va contenir toutes les représentations graphiques des réglages pour un objet sélectionné.
StackPieces : Attribut de type StackPane qui va contenir toutes les représentation graphiques de toutes les pièces de notre Smart Home.

Explication des méthodes

void actionSurObjet(int idObjet) : Méthode qui prend en argument l'identifiant de l'objet et qui permet d'appliquer une action sur cet objet (exemple : allumer un objet connecté).
void reglerObjet(int idObjet, int valeurModifiee) : Méthode qui permet d'appliquer une modification(nouveau réglage) sur un paramètre d'un objet après une modification au niveau de l'interface,'valeureModifiee'sera changé. Elle identifie le paramètre à modifier au niveau de l'interface ce qui permettra à la méthode de récupérer la modification et de la mettre a jour les paramètres de l'objet.
void ajouterObjet() : Méthode qui permet de rajouter un nouvel objet au niveau du gestionnaire de données.
void ajouterPiece() : Méthode qui permet de rajouter une nouvelle pièce au niveau du gestionnaire de données .
void construire() : Méthode qui permet de récupérer les objets et les pièces de la base de données puis reconstruire l'interface selon la dernière sauvegarde.

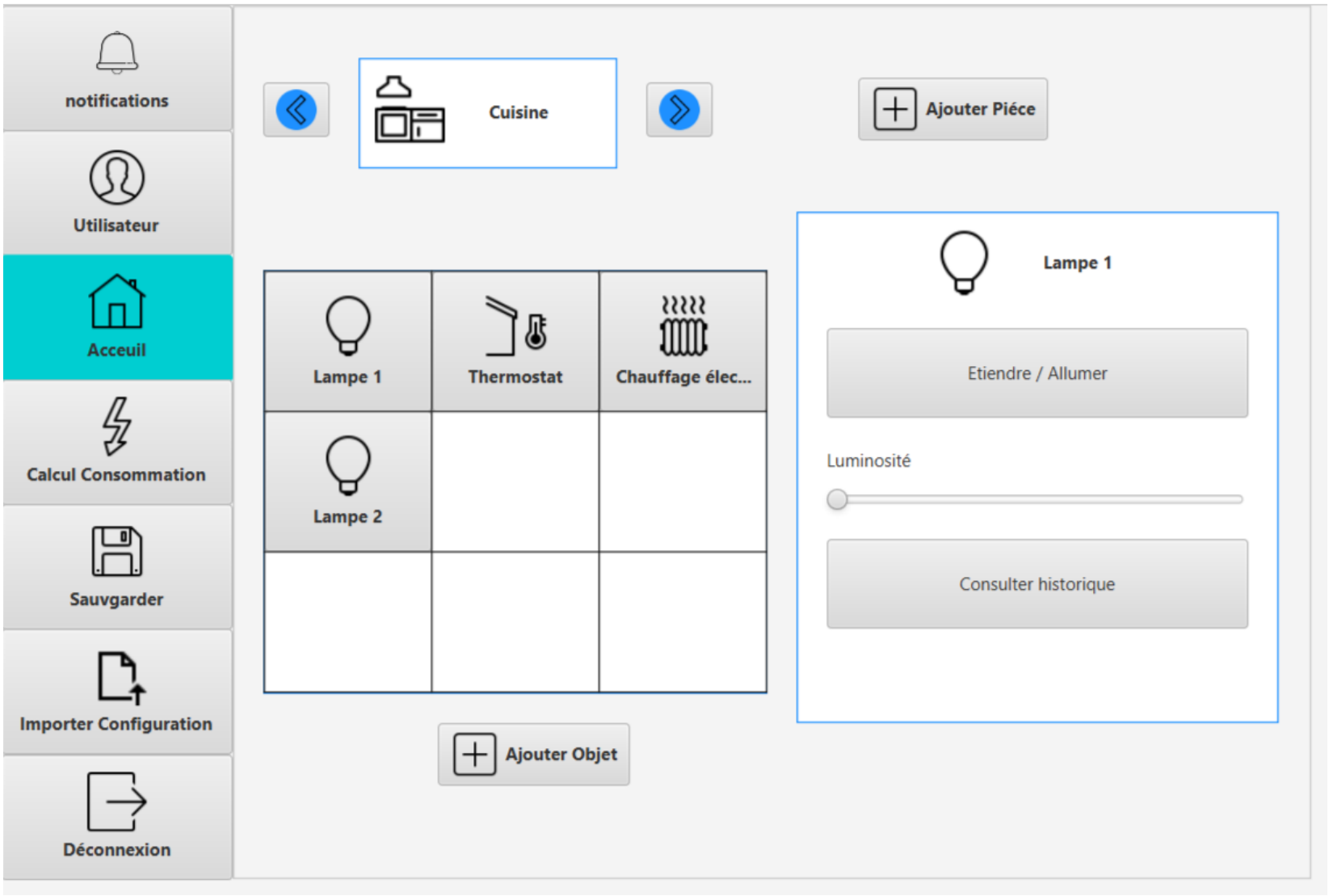


FIGURE 5 – Maquette représentative des fonctionnalités de la classe GestionObjetControlleur

3.4.4 La classe FenetresSecondaires

Cette classe va permettre d'afficher les fenêtres secondaires et de faire communiquer des données provenant de ces fenêtres vers la fenêtre principale "GestionObjet".

```
public class FenetresSecondaires {
public ArrayList<String> afficherAjoutPiece() ;
public ArrayList<Object> afficherAjoutObjet() ;
}
```

Explication des méthodes

ArrayList<String> afficherAjoutPiece() : Méthode qui permet d'afficher la fenêtre d'ajout de pièces elle permet à l'utilisateur de spécifier le type et le nom de la pièce qui sont des String, la méthode créé une ArrayList<String> qui contiendra les deux champs correspondants aux données de la nouvelle pièce et retourne donc un $<ArrayList<String>$ comme valeur de retour.
Elle participe à la réalisation de la fonctionnalité (4) du module interface graphique :"Option d'ajout d'un objet connecté au système" **ArrayList<Object> afficherAjoutObjet() :** Méthode qui permet d'afficher la fenêtre d'ajout d'objets, comme nos objets connectés ont différents types d'attributs, et comme nous avons la possibilités de définir un format propre(formulaire de saisie), on stockera les différents champs représentants les données du nouvel objet dans la $ArrayList<String>$et retourner un ArrayList<Object> et retourner la liste d'attributs à la méthode ajouterObjet() présente dans GestionObjetsControlleur.

3.4.5 La classe CalculControlleur

Cette classe va nous permettre de faire des calculs sur nos objets et les afficher sous forme de tableau et de graphe.

```
import javafx.scene.control.TableView;
import javafx.scene.chart.LineChart;
import javafx.scene.control.DatePicker;

public class CalculControlleur {
private TableView TableauCalcul;
private LineChart GrapheCalcul;
private DatePicker dateDeb;
private DatePicker dateFin;

public Resultat calculerUnique(int idObjet,Date dateDeb, Date dateFin);
public ArrayList<Resultat> calculerPlusieurs(int [] idObjets,Date dateDeb, Date dateFin);
public void afficherGraphe(int idObjet,Date dateDeb, Date dateFin);
public void afficherTableau(int [] idObjets,Date dateDeb, Date dateFin);
}
```

Explication des attributs

TableauCalcul : Attribut de type TableView qui va nous permettre d'afficher les calculs sous forme d'un tableau.
GrapheCalcul : Attribut de type LineChart qui va nous permettre d'afficher les calculs sous forme d'un graphe.
dateDeb : Attribut de type DatePicker qui va permettre à l'utilisateur de choisir la date de début du calcul à faire.
dateFin : Attribut de type DatePicker qui va permettre à l'utilisateur de choisir la date de fin du calcul à faire.

Explication des méthodes

Resultat calculerUnique(int idObjet, Date dateDeb, Date dateFin) : Méthode qui permet de calculer la consommation d'un seul objet pour une durée qui va de la date dateDeb jusqu'à dateFin. Elle retourne un objet de type Resultat
ArrayList<Resultat> calculerPlusieurs(int [] idObjets,Date dateDeb, Date dateFin) : Méthode qui permet de calculer la consommation d'un groupe d'objets dont les identifiants sont connus et cela pour une durée qui va de la date dateDeb jusqu'à dateFin. Elle retourne un objet de type ArrayList<Resultat>
void afficherGraphe(Resultat Res,Date dateDeb, Date dateFin) : Méthode qui permet d'afficher les résultats d'un calcul sous forme d'un graphe.
void afficherTableau(Resultat Res,Date dateDeb, Date dateFin) : Méthode qui permet d'afficher les résultats d'un calcul sous forme d'un tableau.

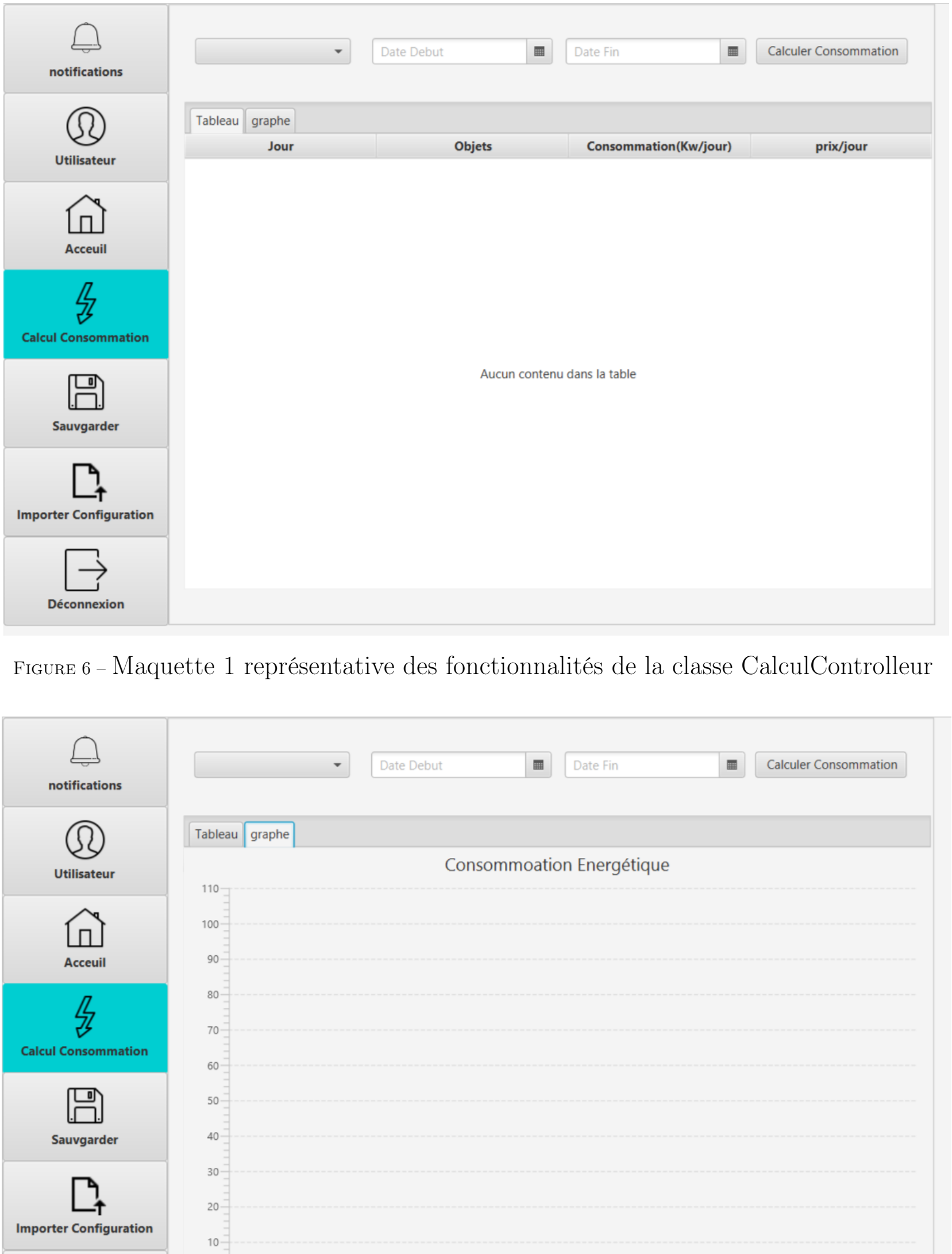


FIGURE 6 – Maquette 1 représentative des fonctionnalités de la classe CalculControlleur

FIGURE 7 – Maquette 2 représentative des fonctionnalités de la classe CalculControlleur

3.4.6 La classe UtilisateurControlleur

Cette classe nous permet de visualiser les données du profil de l'utilisateur(Le nom,le prénom,email..) et offre la possibilité de les modifier.

```
import javafx.scene.control.TextField;

public class UtilisateurControlleur{
private TextField nomField;
private TextField prenomField;
private TextField motDePasseField;

public void changerNom(String nom);
public void changerPrenom(String prenom);
public void changerMotDePasse(String mdp);

public void changerPrixHP(float prixHP);
public void changerPrixHC(float prixHC);

public void changerSeuilConsoHoraire(float seuil);

public void changerHeuresHP1(LocalTime heureDeb,LocalTime heureFin);
public void changerHeuresHP2(LocalTime heureDeb,LocalTime heureFin);


{
```

Explication des attributs

nomField : Attribut de type TextField qui va permettre à l'utilisateur de saisir son nom.
prenomField : Attribut de type TextField qui va permettre à l'utilisateur de saisir son prénom.
motDePasseField : Attribut de type TextField qui va permettre à l'utilisateur de saisir son mot de passe.

Explication des méthodes

void changerNom(String nom) : Méthode qui prend en argument un String et qui permet de changer le nom au niveau de la base de données.
void changerPrenom(String prenom) : Méthode qui prend en argument un String et qui permet de changer le prénom au niveau de la base de données.
void changerMotDePasse(String mdp) : Méthode qui prend en argument un String et qui permet de changer le mot de passe au niveau du gestionnaire de données.

void changerPrixHP(float prixHP) : Méthode qui permet de changer les prix HP par l'utilisateur elle prend en paramètre le prix de type 'float'.
void changerPrixHC(float PrixHC) : Méthode qui permet de changer les prix HC par l'utilisateur elle prend en paramètre le prix de type 'float'.
void changerHeuresHP1(LocalTime heureDeb,LocalTime heureFin) : Méthode qui permet de changer les heures du HP1 par l'utilisateur ils prend en paramètre l'heure de début et l'heure de fin.
void changerHeuresHP2(LocalTime heureDeb,LocalTime heureFin) : Méthode qui permet de changer les heures du HP2 par l'utilisateur ils prend en paramètre l'heure de début et l'heure de fin.

void changerSeuilConsoHoraire(float seuil) : Méthode qui change le seuil de conso horaire de la maison.

3.4.7 La classe IdentificationControlleur

Cette classe contrôle la première fenêtre qui sera affichée au lancement de l'application.Elle va gérer la connexion d'un utilisateur et la création de son compte.

```
import javafx.scene.control.TextField;

public class IdentificationControlleur {
private TextField emailField;
private TextField seConnecterMotDePasseField;

private TextField CreerNomField;
private TextField CreerPrenomField;
private TextField CreerMotDePasseField;
private TextField CreerEmailField;

public void seConnecter(String identifiant,String motDePasse);
public void creerCompte(String nom,String prenom, String motDePasse, String email);
}
```

Explication des attributs

emailField : Attribut de type TextField qui va permettre à l'utilisateur de saisir son e-mail pour se connecter à son compte.
seConnecterMotDePasseField : Attribut de type TextField qui va permettre à l'utilisateur de saisir son mot de passe pour se connecter à son compte.
CreerNomField : Attribut de type TextField qui va permettre à l'utilisateur de saisir son nom lors de la création du compte.
CreerPrenomField : Attribut de type TextField qui va permettre à l'utilisateur de saisir son prénom lors de la création du compte.
CreerMotDePasseField : Attribut de type TextField qui va permettre à l'utilisateur de saisir son mot de passe lors de la création du compte.
CreerEmailField : Attribut de type TextField qui va permettre à l'utilisateur de saisir son émail lors de la création du compte.

Explication des méthodes

void seConnecter(String identifiant,String motDePasse) : Méthode qui permet de vérifier la conformité des identifiants saisis en actionnant la méthode *verifierIdentifiants(String mailUtilisateur, String mdp)* présente dans la classe BDDSingleton.
void creerCompte(String nom,String prenom, String motDePasse, String email) : Méthode qui permet d'ajouter le nouveau compte dans la base de données en appelant l'instance de BDDSingleton et en actionnant la méthode decisionRequete(String requete).

3.4.8 La classe HistoriqueControlleur

Cette classe va permettre à l'utilisateur de calculer la consommation énergétique et de télécharger l'historique de consommation d'un objet connecté.Elle se charge donc de la fonctionnalité (5) du module interface graphique :"Donner une vision globale/détaillée sur la consommation énergétique de la maison (historique de consommation)".

```
import javafx.scene.control.TableView;
import javafx.scene.control.DatePicker;

public class HistoriqueControlleur {
private DatePicker dateDebHistorique;
private DatePicker dateFinHistorique;
private TableView TableauHistorique;
private ResultSet ResultatHistorique;

public ResultSet consulterHistorique(Date dateDeb, Date dateFin);
public void afficherHistorique(ResultSet historique);
public void telechargerHistorique(Date dateDeb,Date dateFin);
}
```

Explication des attributs

dateDebHistorique : Attribut qui va permettre à l'utilisateur de choisir la date de début de l'historique désiré.
dateFinHistorique : Attribut qui va permettre à l'utilisateur de choisir la date de Fin de l'historique désiré.
TableauHistorique : Attribut qui va nous permettre d'afficher l'historique sous forme d'un tableau.
ResultatHistorique : Attribut qui va stocker l'historique consulté par l'utilisateur.

Explication des méthodes

ResultSet consulterHistorique(Date dateDeb, Date dateFin) : Méthode qui permet de récupérer l'historique souhaité de la base de données sous forme de ResultSet.
void afficherHistorique(ResultSet historique) : Méthode qui permet d'afficher l'historique d'un objet sous forme de tableau.
void telechargerHistorique(Date dateDeb,Date dateFin) : Méthode qui permet de créer le fichier contenant l'historique d'un objet à partir du resultSet 'ResultatHistorique'.

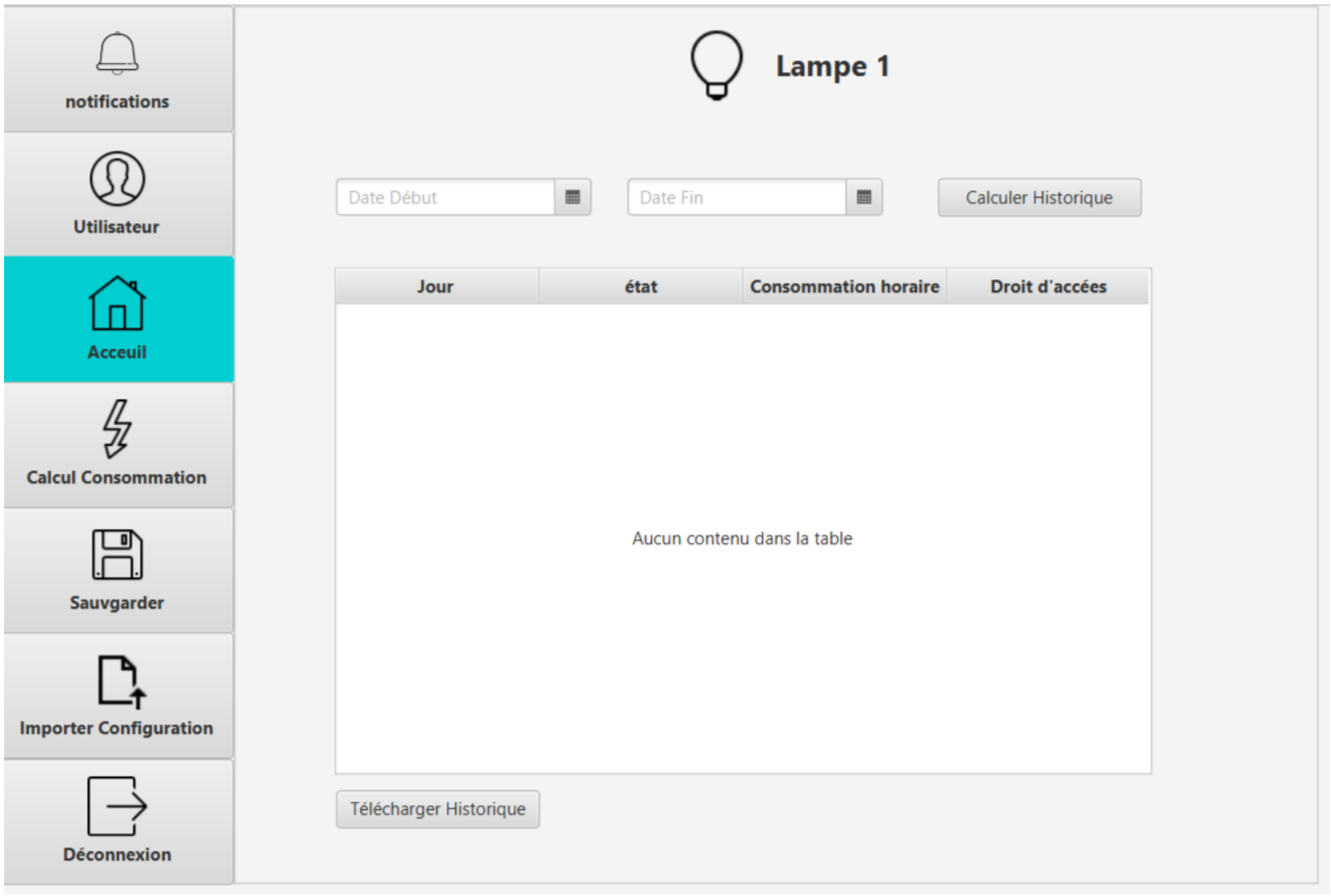


FIGURE 8 – Maquette représentative des fonctionnalités de la classe HistoriqueControlleur

3.4.9 La classe NotificationControlleur

Cette classe gère les notifications envoyées par le système au niveau de l'interface.
Elle fait donc la fonctionnalité (7) du module interface graphique : Fenêtre de visualisation des notifications et des traitements du système de gestion.

```
public class NotificationControlleur {
private static ArrayList<String> ListeNotifications;
public static void ajouterNotification(int type,String notificationText);
public void afficherNotifications();
public void viderHistoriqueNotifs();
}
```

Explication des attributs

ListeNotifications : Attribut qui représente la liste des notifications ordonnées selon le temps d'apparition.

Explication des méthodes

void ajouterNotification(int type,String notificationText) : Méthode qui permet d'ajouter la notifications passée en paramètre dans 'ListeNotifications'. La méthode est appelé au niveau des autres modules du logiciel afin d'afficher à l'interface les différentes décisions prises automatiquement dans le programme.
void afficherNotifications() : Méthode qui permet d'afficher toutes les notifications au niveau de l'interface
void viderHistoriqueNotifs() : Méthode qui permet de supprimer les notifications au niveau de la classe et au niveau de l'interface

4 Conclusion

La conception de notre cahier de spécifications nous a permis d'avoir une visualisation beaucoup plus précise du travail qui nous reste à faire dans la dernière partie de la réalisation du projet soit la partie développement. cela nous a aussi permis de confirmer la fiabilité de notre organigramme et des informations qui circulent entre les différents modules qui le composent. Concernant notre choix du langage de programmation : on trouve que le langage JAVA convient finalement très bien au développement de notre projet.
Pour le développement de notre application,on envisage de créer un dépôt gitHub où on va stocker notre code,on continuera à nous réunir tous ensemble deux fois par semaine pour voir l'avancement de l'implémentation de chaque module.Nous estimons cette méthode correcte puisqu'elle nous a permit d'être dans les temps pour le cahier de charges et le cahier de spécifications du projet.

5 Glossaire

- Réf(1) : https://www.sqlite.org/src/doc/trunk/README.md

- Réf(2) : https://www.developpez.com/actu/94614/Un-developpeur-evoque-cinq-raisons-pour-vous
#:~:text=L%27un%20des%20avantages%20de,le%20cas%20d%27application%20embarqu%C3%A9es

- Réf(3) : https://fr.wikipedia.org/wiki/SQLite