Université Versailles Saint Quentin-En-Yvelines

UFR des sciences
Département Informatique

Rapport finale

Simulation d'un système de carburant d'un avion de chasse

Présenté par:

M^{r}	**BIZIBANDOKI**	**Louange.**
M^{ell}	**LOUIBA**	**Siham .**

Année Universitaire 2014 – 2015

Table des matières

Table des figures

1 Introduction

Le système de carburant gère les opérations de ravitaillement et de vidange, ainsi que les flux de carburant entre le moteur et les réservoirs pendant le vol. Il permet aussi de transférer le carburant d'un réservoir à l'autre.
Le but de notre projet et de d'implémenter un simulateur pour ce système afin que les pilotes puissent s'entrainer avant d'aller sur le terrain. Pour ce faire nous avons opté pour le langage de programmation C++ pour divers raisons tel que le fait qu'il soit un langage hybride donc offre plus de fluidité dans la programmation.
Nous avons était deux personne pour la réalisation et l'élaboration de ce projet.

Notre projet est divisé en trois modules: Interface, Modèle et Base de données.

2 Architecture du logiciel :

Notre projet est composé de trois modules principaux basés sur une architecture MVC (Modèle – Vue – Contrôleur), celle-ci est donc modulaire et rend le logiciel facilement maintenable car la logique du code est séparée en trois parties que l'on retrouve dans des fichiers différents et dont la gestion interne est entièrement autonome.

2.1 Le Modèle

Cette partie gère les données de l'application stockées dans des fichiers ou entrées par l'utilisateur. Elle assure aussi la cohésion entre les différents composants du système carburant que nous allons décrire dans les lignes qui suivent. Ainsi donc le modèle est composé de deux parties majeures : le système carburant et la base de données.

2.1.1 Le système carburant

C'est le module du programme qui modélise sous forme d'un modèle calculatoire de données compréhensible par l'ordinateur le système carburant d'un avion de chasse, ses états et ses fonctions. Il s'agit donc de la représentation sous forme de classes des composants comme les réservoirs, les moteurs, les pompes, les vannes... Toutes ces classes communiquent entre elles fournissant ainsi les fonctionnalités d'un vrai système d'alimentation selon le modèle qui a été demandé.

Se référant au paradigme de la programmation objet, nous avons pensé le Système Carburant comme étant lui-même un objet dit « central » composé de réservoirs, de mo-

teurs, de vannes, de pompes de canaux d'alimentation reliant les réservoirs ou ces derniers aux moteurs et contrôlés par des vannes. La classe **SystemeCarburant** permet donc de modéliser notre « objet central ».En suivant donc le même raisonnement, nous avons établis une hiérarchie entre les différents composants :

- Un réservoir contient deux pompes (une pompe primaire et une pompe de secours). Cf. classe **Reservoir**.
- Les moteurs, les vannes VT (VT12 et VT23) sont des objets qui ne sont pas contenus ni contenant d'autres composants.
- Les connexions entre les réservoirs et les moteurs sont gérées par la classe **VanneMatrice** qui contient (entre autre) les vannes V (V12, V13, V23) et une matrice d'adjacence permettant de connaître quel réservoir alimente quel moteur.

Toutefois si un utilisateur souhaite garder un historique de ses simulations, le module suivant permet de le faire.

2.2 La base de données

Ce module gère l'enregistrement dans des fichiers et le chargement à partir des mêmes fichiers des données stockées par les utilisateurs. En effet ces fonctionnalités sont indispensables pour assurer efficacement les services d'authentification des utilisateurs et la gestion de l'historique de ses exercices.

Pour ces deux modules, un listing complet des méthodes de ces classes est donné dans la suite du rapport.

3 La vue

Cette partie de l'architecture intègre un module important de l'application : l'interface graphique. L'interface graphique assure l'interaction entre l'utilisateur et le modèle qu'il ne voit pas. C'est la vitrine qui représente en temps réel l'état du Système Carburant et qui permet à l'utilisateur de lui apporter des modifications. Notre interface graphique a été développée à l'aide la bibliothèque graphique Qt et possède deux fenêtres principales qui s'affichent au moment de la simulation. Tout d'abord la première fenêtre (image 1) affiche l'état du Système et de tous ses composants (état des vannes, des pompes, des pompes et des réservoirs, flux du carburant...). Elle permet également via le mécanisme des widgets

de choisir une configuration pour le Système (capacité des réservoirs, consommation des moteurs,...). Enfin elle permet la gestion de l'état de la simulation (mettre en pause, arrêter, refaire), la gestion des connexions et des services associés tels que la création d'un compte, l'affichage des notes de l'utilisateur, le choix des challenges (exercices) à réaliser.

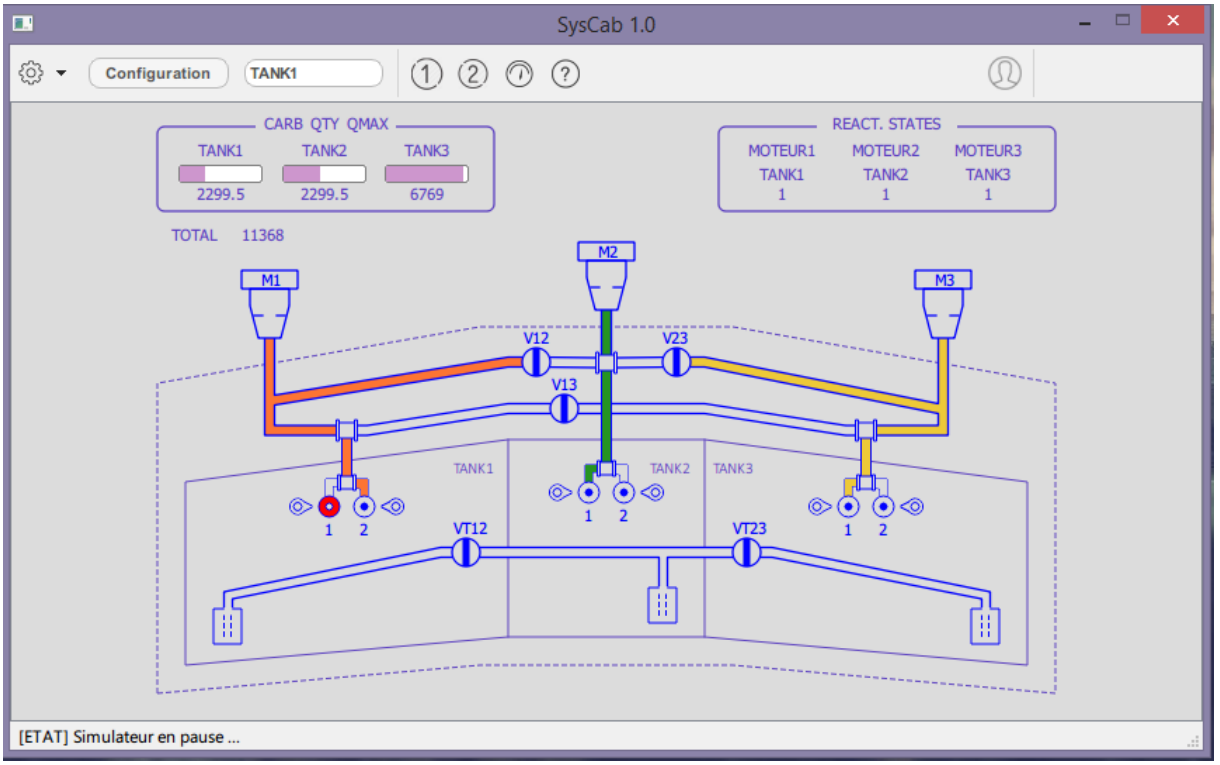

Figure 1: Fenêtre principale.

La seconde fenêtre (image 2) représente le tableau de bord. Celui-ci contient les boutons gérant l'ouverture/fermeture des pompes et des vannes. Elle ne s'affiche qu'en cas de simulation active. Chaque action reportée entraine une modification du modèle.

Figure 2: Tableau de bord.

Nous n'avons pas dans ce rapport présenté le listing des classes du module Interface graphique.

Nous avons précédemment dit qua chaque module de notre application se gérait de façon autonome. Ainsi le fonctionnement ou non du modèle ne dépend pas de l'interface graphique par exemple et inversement. Cependant pour assurer la communication entre les deux modules, nous avons la troisième partie de l'architecture qui permet de gérer cela.

4 Contrôleur

Afin de s'assurer que les actions de l'utilisateur sont affectées au modèle et que les informations cohérentes de ce dernier sont affichées par l'interface, le contrôleur assure l'intermédiaire entre ces deux parties. Ainsi donc une requête de l'utilisateur est captée par le contrôleur et transmise au modèle après avoir été vérifiée ; les modèle répond donc en modifiant son comportement et en signalant à la vue son nouvel état.

La classe **SysCabClassePrincipale** est le point d'entrée de notre programme. Celle-ci contient des instances du modèle (Système Carburant et Base de données) et de **l'interface graphique**. Enfin elle gère les processus de signaux et slots (concept de Qt et assurant la communication entre les différents composants graphiques).

5 Listing complet des fonctions de l'application

5.1 Classe Moteur

La classe Moteur est la classe représentant un moteur du Système Carburant. Tout moteur se construit en fournissant son nom et sa consommation de carburant par unité de temps (5ms).

5.1.1 Propriétés :

__ **nom : string :** Cette propriété contient le nom du moteur. Elle est utilisée par l'application et les utilisateurs pour identifier, trouver ou communiquer avec l'objet. Elle ne contient pas de valeur par défaut et est requise pour l'instanciation d'un objet de la classe.
Fonction d'accès : string getNom() const et void setNom(string).

__ **conso : double :** Cette propriété contient la valeur de la consommation du moteur en carburant pendant un tour d'alimentation (Cf. fonction void consommation() de

la classe SystemeCarburant). Elle ne contient pas de valeur par défaut et est requise pour l'instanciation d'un objet de la classe.
Fonctions d'accès : double getConso() const et void setConso(double).

__ **bObserve : bool :** Cette propriété contrôle si le moteur est actuellement alimenté par un des réservoirs. Définir cette propriété à true annonce au système que le moteur a été alimenté lors de la dernière consommation. Par défaut sa valeur est à false.
Fonction d'accès : bool isAlimente() const et void updateCanalResObserver(bool).

__ **controlePanne : bool :**Cette propriété contrôle si le moteur a subi une panne (arrêt d'alimentation). Par défaut à false, elle est définie à true si une panne est intervenue dans le système.
Fonction d'accès : void activeControlePanne().

__ **canalObserver: GraphicCanalObserver** Cette propriété contient l'adresse du canal graphic reliant le moteur. Elle permet au moteur de signaler à un objet GraphicCanalObserver (du module Interface Graphique) ses changements d'états afin de mettre à jour l'affichage. Par défaut, cette propriété est vide.
Fonction d'accès : GraphicCanalObserver canalObserver()const, et void addCanalObserver(GraphicCanalOserver).

__ **reservoirObserver : Reservoir** Cette propriété contient l'adresse du réservoir qui l'alimente en carburant. Par défaut à NULL, cette propriété change chaque fois qu'un nouveau réservoir alimente un moteur.
Fonction d'accès : Reservoir reservoirObserver() const,
void updateResObserver(Reservoir) et deleteResObserver().

__ **controlePanne : bool** Cette propriété alerte le moteur d'une interruption d'alimentation intervenue suite à une panne dans le système, elle s'active à true et le moteur n'est plus alimenté. Elle est par défaut à false.
Fonction d'accès : void activeControlePanne().

5.1.2 Fonctions membres :

Moteur ::Moteur (string nom, double conso Construit un moteur avec son nom et sa consommation. Ces deux paramètres sont obligatoires.

Moteur ::M̃oteur() Détruit l'objet moteur.

void Moteur ::affiche() Affiche l'état actuel du moteur dans une sortie standard.

string Moteur ::getResObserverName() const Renvoie s'il n'est pas vide le nom du réservoir qui l'alimente.

Void Moteur ::initialise (Reservoir obs, double conso = -1) Cette fonction permet de (ré)initialiser les propriétés de l'objet ; cependant l'identité reste la même. Si conso < 0, alors la valeur actuelle de _ conso du moteur ne sera pas modifiée.

5.2 Classe Pompe

Cette classe représente une une pompe du système. C'est une classe abstraite héritée par les classes PompePrimaire et PompeSecours.

5.2.1 Propriétés :

_ nom: string : Cette propriété contient le nom de la pompe permettant à l'utilisateur de l'identifier. Elle n'a pas de valeur par défaut et est initialisé grâce au constructeur. **Fonction d'accès :** string getNom() const et void setNom(string nom).

_ panne : bool Cette propriété permet de contrôler l'état de la pompe. Si elle est à true, la pompe est donc en panne. Par défaut elle est initialisée à false. **Fonctions d'accès :** bool isPanne() const et void setPanne(bool panne).

5.2.2 Fonctions membres :

Pompe ::Pompe (string nom) : Construit un objet Pompe avec son nom.

Pompe ::$\tilde{\text{P}}$ompe() Détruit un objet pompe.

Virtual void Pompe ::affiche() const = 0 Affiche l'état de la pompe. Fonction virtuelle pure redéfinie dans les classes filles.

5.3 Classe PompePrimaire :

Cette classe représente la pompe primaire d'un réservoir du système carburant. Elle hérite de la classe Pompe. Elle est construite par invocation du constructeur de sa classe mère. Une pompe primaire n'a que deux états (panne ou fonctionnel), elle est par défaut en marche.

5.3.1 Fonctions membres :

void PompePrimaire ::affiche() const : C'est une méthode constante qui ne retourne rien mais qui permet l'affichage de l'état de la pompe.

5.3.2 Classe PompeSecours :

Cette classe représente la pompe de secours d'un réservoir du système carburant. Elle hérite de la classe Pompe.

5.3.3 Propriétés

__ **marche : bool** : Cette propriété définit si l'état actuel de la pompe. Si la propriété est vraie alors la pompe est en marche. Par défaut elle est à false. Cependant, cette propriété ne peut qu'être définie à true que dans le cas où la pompe n'est pas en panne.
Fonction d'accès : bool getMarche() et void setMarche(bool marche).

5.3.4 Fonction membres :

PompeSecours ::PompeSecours(string nom) Construit une pompe de secours avec le nom en invoquant le constructeur de sa classe mère.

PompeSecours ::P̃ompeSecours() Détruit l'objet de la classe PompeSecours

PompeSecours ::setPanne (bool panne) Redéfinition de la méthode de la classe mère. Elle permet d'arrêter une pompe si elle est en panne.

5.4 Classe réservoir :

Cette classe représente le réservoir d'un système carburant.

5.4.1 Propriétés :

__ **nom : string** Cette propriété le nom du réservoir permettant au système et à l'utilisateur de l'identifier. Elle est initialisée via le constructeur.
Fonction d'accès : string getNom() const, void setNom(string nom).

__ **capacite : double** Cette propriété contient la capacité d'un réservoir, c'est-à-dire la quantité maximale de carburant qu'il peut contenir. Elle est initialisée à la construction de l'objet.
Fonctions d'accès : double getCapacite() const, void setCapacite(double capacite).

__ **qteCarburant : double** Cette propriété contient la quantité de carburant présente dans le réservoir. Par défaut le réservoir est remplie et donc sa valeur est égale à celle de la capacité du réservoir.

Fonctions d'accès : getQteCarburant() const, double setQteCarburant(double qte). Cette dernière renvoie la valeur de la quantité de carburant en excès.

_ vide : bool : Cette propriété permet de savoir l'état du réservoir par rapport à la quantité de carburant. Si _ vide est true alors le réservoir est vide. Par défaut, elle est initialisée à false.
Fonction d'accès : bool isVide()const, void setVide(bool vide).

_ pompePrimaire : PompePrimaire Cette propriété contient la pompe primaire du réservoir. Celle-ci est initialisée dans le constructeur avec le nom «1».
Fonction d'accès : PompePrimaire getPompePrimaire() const, void setPompePrimaire (PompePrimaire pp).

_ pompeSecours : PompePrimaire : Cette propriété contient la pompe de secours du réservoir. Celle-ci est initialisée dans le constructeur avec le nom «2».
Fonctions d'accès : : PompeSecours getPompeSecours() const, void setPompeSecours (PompeSecours pp).

_ pompePanne : bool : Cette propriété contrôle si une des pompes du réservoir est en panne. Elle est à true si une panne de pompe non résolue est intervenue dans le réservoir. Elle est à false par défaut et lorsque la panne est résolue. Elle n'est consultable que par le réservoir mais peut être modifié de l'extérieur.
Fonctions d'accès : setPompePanne(bool b).

_ carbPanne : bool Cette propriété contrôle si une panne de « vidange » du réservoir et non résolu est intervenue, dans ce cas elle est à true. Elle est à false par défaut et lorsque la panne est résolue. Elle n'est consultable que par le réservoir mais peut être modifié de l'extérieur.
Fonction d'accès : void setCarbPanne(bool b).

_ cptDecision : int Cette propriété indique le nombre de bonnes décisions à la résolution des pannes du réservoir courant que l'utilisateur a prises. Par défaut elle est initialisée à 0. Chaque que l'utilisateur prend une bonne décision (selon les règles établies) pour résoudre une panne, on incrémente ce compteur. Sa modification est donc interne.
Fonction d'accès : Cette propriété indique le nombre de bonnes décisions à la résolution des pannes du réservoir courant que l'utilisateur a prises. Par défaut elle est initialisée à 0. Chaque que l'utilisateur prend une bonne décision (selon les règles

établies) pour résoudre une panne, on incrémente ce compteur. Sa modification est donc interne.

__ **cptPanneResolus : int** Cette propriété indique le nombre de pannes du réservoir que l'utilisateur a résolu pendant une simulation. Sa valeur par défaut est 0. Ainsi chaque fois que l'utilisateur résous une panne intervenue dans le réservoir, on incrémente ce compteur que la décision soit la meilleure ou pas.
Fonctions d'accès : getCptResolutions() const.

__ **cptPannes : int** Cette propriété indique le nombre total de pannes intervenues dans le réservoir lors d'un simulation. Sa valeur par défaut est 0.
Fonction d'accès : int getCptPannes() const.

__ **moteurSupId : int** : Cette propriété contient le nom du moteur supplémentaire que le réservoir alimente. Sa valeur est donc initialisée si le réservoir alimente un second moteur en dehors de celui qu'il alimente par défaut.
Fonction d'accès : string moteurSupId() const, void updateMoteurSup(string nom).

__ **resObserver : GraphicReservoir** Cette propriété contient l'adresse du réservoir graphique (modèle dessiné) que le réservoir du modèle va observer. L'initialisation de celle-ci se fait au moment de la classe principale. On configure chaque réservoir et moteur de manière à ce qu'il observe une composante graphique.
Fonctions d'accès : GraphicReservoir resObserver() const, void addGraphicReservoir(GraphicReservoir pp)

5.5 Fonctions membres :

Reservoir ::Reservoir(string nom, double capacite) : Construit un objet réservoir avec son nom et sa capacité.

Reservoir ::R̃eservoir() : Détruit l'objet de la classe.

Void Reservoir ::viderCarburant() : Cette fonction ne prend aucun paramètres et ne renvoie rien. Elle vide le carburant du réservoir.

Void Reservoir ::demarrerPompeSecours() Permet de démarrer la pompe de secours du réservoir.

Void Reservoir ::fermerPompeSecours() Permet de fermer la pompe de secours du réservoir.

Void Reservoir ::setPannePompePrimaire() Permet de générer la panne de la pompe primaire du réservoir.

Void Reservoir ::setPannePompeSecours() Permet de générer la panne de la pompe de secours du réservoir/

Bool Reservoir ::isDiffuseCarburant() Permet de savoir si le réservoir peut diffuser du carburant selon son état (vide ou non) ou celui de ses pompes (en panne ou non).

Bool Reservoir ::isPompesDiffusent() Permet de savoir si les deux pompes sont simultanément en état de diffuser du carburant.

Bool Reservoir ::update() Cette fonction permet à chaque composant du réservoir de mettre à jour le composant graphique qu'il observe en envoyant son nouvel état.

Void Reservoir ::initialise(int capacite = 1) Initialise les propriétés du réservoir à défaut sans modifier son identité. Si la valeur en entrée est supérieure à 1 alors le réservoir est initialisée avec sa nouvelle capacité, sinon on ne change pas celle-ci.

Void Reservoir ::controlPanneResSignal(const string& sMoteur, const string& sNouvRes) Cette fonction permet de signaler au réservoir que la panne courante a été résolue. Elle prend en entrée le nom du moteur signalant la résolution et le nom du nouveau réservoir qui va l'alimenter

5.6 Classe Vanne :

Cette classe représente toute vanne du système carburant.

5.6.1 Propretés :

__ **nom : string** Cette propriété contient le nom permettant d'identifier la vanne.
Fonctions d'accès : string getNom() const, void setNom(string nom).

__ **ouvert : bool** Cette propriété permet de contrôler l'état de la vanne. A true, la vanne est ouverte sinon elle est fermée. Une vanne est par défaut ouvert.
Fonction d'accès : bool isOuvert() const ; setOuvert(bool ouvert).

__ **vanneObserver : GraphicVanne** Cette propriété contient l'adresse de la vanne graphique qu'il observe. Par défaut elle est vide.
Fonction d'accès : GraphicVanne* vannObserver() const, void addVanneObserver (GraphicVanne gv).

5.6.2 Fonction membres :

Vanne ::Vanne(string nom) : Construit une vanne avec le nom en paramètre.

Vanne ::$\tilde{\text{V}}$anne() : Détruit l'objet de la classe vanne.

Void Vanne ::affiche(): Affiche l'état de la vanne dans le terminal.

Void Vanne ::update() : Met à jour l'état de la vanne graphique.

Void Vanne ::updateColor(const QBrush& br, bool b) : Modifie la couleur des canaux qui passent par la vanne. Si b est vrai alors on permet la modification avec la nouvelle couleur en paramètre br, sinon on utilise la couleur par défaut.

5.7 Classe VanneMatrice

Cette classe permet de gérer les connexions entre les moteurs et les réservoirs.

5.7.1 Propriétés :

- **_ vannes : vector<Vanne>** : Cette propriété contient les vannes qui servent de point de connexions entre réservoirs et moteurs. Par défaut, elle contient trois (3) vannes.
  Fonctions d'accès : : vector<Vanne> vannes() const, void setVannes(const vector<Vanne>& vannes), Vanne& getVanne(int i).

  _ matrice : int[][] Cette propriété est une matrice d'adjacence permettant de dire si la vanne connectant un réservoir i et un moteur j est ouverte ou fermée. Par défaut, elle est une matrice 3 3.
  Fonction d'accès : int getEntree(int i, int j), void setEntree(int l, int col, int value).

5.8 Fonctions membres :

VanneMatrice ::VanneMatrice() Construit par défaut un objet vanne matrice.

VanneMatrice ::$\tilde{\text{V}}$anneMatrice() Détruit un objet de la classe.

Void VanneMatrice ::ouvrirVanne(int i) Ouvre la vanne se trouvant à l'indice i du tableau _ vannes.

Void VanneMatrice ::fermerVanne(int i) Ferme la vanne se trouvant à l'indice i du tableau _ vannes.

Void VanneMatrice ::afficher() Affiche l'état de l'objet et de la matrice dans le terminal.

5.9 Classe SystemeCarburant

Cette classe est la classe qui représente le système carburant dans sa globalité avec ses composants et les fonctions permettant de les gérer.

5.9.1 Propriétés :

_ reservoirs : vector<Reservoir> : Cette propriété contient le tableau de tous les réservoirs du système.
Fonction d'accès : Reservoir& getReservoir(int i).

_ moteurs : vector<Moteur> : Cette propriété contient le tableau de tous les moteurs du système.
Fonction d'accès : Moteur& getMoteur(int i).

_ vanneRes : vector<Vanne> : Cette propriété contient le tableau des vannes reliant les réservoirs du système.
Fonction d'accès : Vanne& getVanneRes(int i).

_ vanneRM : VanneMatrice : Cette propriété contient l'objet vanneMatrice représentant les accès et connexions entre les réservoirs et les vannes.

Fonction d'accès : VanneMatrice& getVanneRM().

5.9.2 Fonctions membres :

SystemeCarburant ::SystemeCarburant() : Un constructeur par défaut qui contruit un objet selon le modèle du problème géré.

SystemeCarburant ::S̃ystemeCarburant() : Détruit un objet SystemeCarburant.

void SystemeCarburant ::ouvrirVanneRes(int i) : Ouvre la vanne à l'indice i du tableau de vanne.

void SystemeCarburant ::fermerVanneRes(int i) : Ferme la vanne à l'indice i du tableau de vanne.

void SystemeCarburant ::ouvrirVanneRM(int i) : Ouvre la vanne à l'indice i du tableau de vanne de l'objet _ vanneRM.

void SystemeCarburant ::fermerVanneRM(int i) : Ferme la vanne à l'indice i du tableau de vanne de l'objet _ vanneRM.

void SystemeCarburant ::ouvrirPsecours(int i) : Ouvre la pompe de secours de réservoir à l'indice i du tableau de réservoirs.

void SystemeCarburant ::fermerPsecours(int i) : Ferme la pompe de secours de réservoir à l'indice i du tableau de réservoirs.

void SystemeCarburant ::viderReservoir(int i) : Vide le réservoir à l'indice i du tableau de réservoirs.

void SystemeCarburant ::pannePprimaire(int i) : Génère la panne de la pompe primaire du réservoir à l'indice i du tableau de reservoirs.

void SystemeCarburant ::pannePsecours(int i) : Génère la panne de la pompe de secours du réservoir à l'indice i du tableau de reservoirs.

bool SystemeCarburant ::isValidReservoirIndex(int i) : Vérifie que l'indice i est entre les bornes du tableau de réservoirs.

bool SystemeCarburant ::isValidMoteurIndex(int i) : Vérifie que l'indice i est entre les bornes du tableau de moteur.

bool SystemeCarburant ::isValidVanneResIndex(int i) : Vérifie que l'indice i est entre les bornes du tableau de vannes.

Bool SystemeCarburant ::isReservoirsVide() : Vérifie si tous les réservoirs sont vides.

Bool SystemeCarburant ::consommation() : Cette fonction est l'une des principales de la classe. En effet elle déduit dans tous les réservoirs la quantité de carburant en fonction de la consommation des moteurs qu'ils alimentent. Si tous les réservoirs sont vides alors la consommation n'est pas possible. Elle renvoie un booléen si la consommation a pu être effectuée.

Bool SystemeCarburant ::transfertCarburant() : Assure le transfert du carburant entre réservoirs. Si la vanne reliant deux réservoirs est active alors on équilibre leurs niveaux de carburant.

int getControlDecisions() : Renvoie le nombre total de bonnes résolutions prises par l'utilisateur pour résoudre les pannes. (Cf. classe Reservoir)

int getControlResolutions() : Renvoie le nombre total de pannes résolus. (Cf. classe Reservoir)

int getControlPannes() : Renvoie le nombre total de pannes intervenues dans le système.

void initialise(double r1 = -1, double r2 = -1, double r3 = -1, double m1 = -1, double m2 = -1, double m3 = -1) : Initialise toutes les valeurs composants du modèle ainsi que leur état. Prend en paramètres la capacité des réservoirs (trois premiers paramètres) et la consommation des moteurs (trois suivants). Par défaut les paramètres valent -1. Ce qui veut dire qu'on réinitialise la dernière configuration connue.

void initExoConfig(vector<int> vconf) : Appelle la fonction initialise(double, double, double, double, double, double) pour initialiser le modèle selon la configuration du paramètre vconf. Le paramètre vconf est un tableau de valeurs.

void resetModel() : Réinitialise tout le modèle avec la dernière configuration connue. Cette classe possède également des fonctions de mises à jour permettant de mettre à jour l'interface graphique (affichage de la quantité de carburant, relations moteurs-réservoirs...). Pour cela la classe observe (contient l'adresse de cet élément) un objet GraphicEtatRM (classe du module Interface graphique).

5.10 Bases :

5.10.1 BaseExercices

Cette classe nous permet de stocker les exercices à effectuer lors de la simulation, elle comprend plusieurs méthodes.
getExo(int i) const : Est un getter qui nous permet de récupérer le numéro d'un exercice.
chargerExercices() : Est une méthode qui nous permet de passer d'un exercice à un autre.

5.10.2 BaseUtilisateurs :

bool ajouterUtilisateur(const string& identif, const string& motDePasse) : Est une méthode qui prend en paramètres deux arguments identif qui est l'identifiant de l'utilisateur et motDePasse, cette fonction nous renvoie un booléen vrai dons le cas où

l'utilisateur n'existe pas déjà et nous permettra de l'ajouter faux sinon.

void chargerBase() : Est la méthode qui nous permet de mettre en mémoire le fichier contenant l'ensemble des utilisateurs afin qu'ils puisse être reconnu lors de l'authentification et donc elle ne prend aucun argument en paramètre.

rechercherUtilisateur(const string & id, const string & mdp) : Est une qui ne renvoie un utilisateur, elle prend en paramètre un string id qui est l'identifiant de l'utilisateur à chercher et mdp son mot de passe, elle permet d'effectuer une recherche dans le fichier de la base d'utilisateurs.

6 Conclusion :

Nous somme parvenu à implémenter différentes fonctionnalité du simulateur de système de carburant, présenté une application capable de générer des exercices pour le pilote et lui permet d'enregistrer son historique ainsi que l'ensemble des notes qu'il a obtenu, cependant il n'y accède à son compte qu'après authentification.