

# Rapport Projet IN505

Mahdi Larbi et Mohamed Ramdane Debiane

07 Janvier 2020

Ce rapport est joint avec l'archive du logiciel Simulateur, dans le cadre du projet proposé par Mme Kloul pour le module de Langages Avancés.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Pré-requis . . . . .	2
1.2	Installation . . . . .	2
1.3	Base de donnée des pilotes . . . . .	2
1.4	Documentation . . . . .	3
<b>2</b>	<b>Mise en route</b>	<b>3</b>
<b>3</b>	<b>Simulateur</b>	<b>3</b>
3.1	Core . . . . .	4
3.1.1	Reservoirs . . . . .	4
3.1.2	Pompes . . . . .	4
3.1.3	Moteurs . . . . .	4
3.1.4	Vannes . . . . .	5
3.1.5	Vannes Pompes/Moteurs . . . . .	5
3.1.6	Vannes Reservoir/Reservoir . . . . .	5
3.1.7	Systeme . . . . .	5
3.2	States . . . . .	5
3.2.1	State . . . . .	6
3.2.2	StateManager . . . . .	6
3.2.3	LoginState . . . . .	6
3.2.4	MenuState . . . . .	6
3.2.5	MainGState . . . . .	7
3.3	GUI . . . . .	7
3.3.1	LoginGui . . . . .	7
3.4	MenuGui . . . . .	7
3.4.1	MainGui . . . . .	7
3.5	Utils . . . . .	7

3.5.1	Fichiers de configuration . . . . .	7
3.5.2	Parser . . . . .	8
3.5.3	Pannes . . . . .	9
3.5.4	Notation . . . . .	9
3.5.5	Sauvegarde . . . . .	9
<b>4</b>	<b>Ressources</b>	<b>9</b>
<b>5</b>	<b>Listing</b>	<b>10</b>

# 1 Introduction

## 1.1 Pré-requis

Afin de permettre une compilation, génération de documentaion et execution correcte du projet les packages suivants doivent être installés sur la machine cible:

- Compilateur: GNU/G++.
- Bibliothèque graphique: Qt5
- Base de donnée: SQLite3
- Script de compilation: CMake

## 1.2 Installation

Les sources du projet son disponibles sur GitHub.

La récupération et la compilation se font de la façon suivante :

```
$ git clone https://github.com/Khalimouh/Simulateur.git
$ cd Simulateur/
$ cmake -G "Unix_Makefiles"
$ make release
# Executer le binaire peut se faire de deux fa ons
$ make run
# ou
$ cd bin/Release/ && ./Simulateur
```

## 1.3 Base de donnée des pilotes

Nous avons choisis d’implémenter la base de donnée des pilotes en utilisant la biliothèque Sqlite3, le fichier contenant les tables se trouve à la racine du projet sous le nom: Simulateur.db.

La base contient une seule table conteant le login, le mot de passe, le nom et

prénom du pilote. pour des raisons de sécurité, l'application ne permet pas de modifier ni de faire des insertions dans cette table.

Pour l'instant seul le compte de deux pilotes sont disponible, les developpeurs peuvent cependant ajouter des comptes supplémentaires si besoin est.

## 1.4 Documentation

La documentation est générée avec Doxygen, à chaque appel de la commande make en Release.

## 2 Mise en route

Une fois le programme executé, le pilote doit remplir l'interface de connexion avec ces informations personnelles, une fois la validation effectuée, le pilote arrive sur l'écran de menu de la simulation lui permettant de choisir parmi plusieurs modes de simulation il peut par exemple commencer une nouvelle simulation, en charger une qu'il n'a pas eu le temps de finir, ou commencer une simulation libre selon les 3 niveau de difficulté que nous lui proposons (0 facile à 2 expert), une fois le fichier sélection ou la nouvelle partie libre choisie l'utilisateur se voit afficher la fenetre principale de l'application, avec le tableau de bord et des wid-gets dynamiques qui affichent l'état du système en temps réel.

Il peut alors à l'aide de la souris, ouvrir et fermer des vannes, éteindre ou allumer des pompes et les raccorder avec les différents moteurs selon la situation.

Une fois la simulation terminée correctement: plus de carburant ou limite de temps dépassée, l'utilisateur se voit afficher son score sur l'écran de fin du jeu.

Il lui est aussi possible de sauvegarder sa Simulation en cours et une sauvgarde de la simulation est effectuée à la fin de celle ci.

Les différentes interfaces et états de l'application ainsi que leur implémentation seront expliqués dans les sections suivantes du rapport.

## 3 Simulateur

Nous avons découpé le simulateurs en plusieurs dossiers ayant chacun pour objectif une tache prédéfinie.

Nous avons ainsi 4 modules:

- Core : Regroupe toutes les classes nécessaires à la modélisation du système carburant de l'avion
- States: Contient tous les états de l'application, permet ainsi une transition plus facile entre les différents états

- gui: Contient toutes les interfaces graphiques correspondant au différents états de la simulation
- utils: module utilitaire, contenant un parser de fichier XML ainsi que la définition de la classe panne

### 3.1 Core

Nous avons décidé que pour la modélisation du système carburant chaque pièce du système (Réservoirs, Vannes, Pompes et Moteurs) allaient être représentés par une classe distincte

#### 3.1.1 Réservoirs

Les réservoirs sont caractérisés par des différents attributs, son numéro permettant de le reconnaître, sa capacité maximum, ainsi que son état qui est représenté par une énumération d'états (cf: Doc).

Le réservoir contient aussi les adresses de ces deux pompes ainsi que du moteur qu'il s'occupe d'alimenter en carburant.

La construction de l'objet se fait en passant comme argument un numéro, une capacité et un pointeur sur le moteur (Par défaut T[n] et M[n]).

Plusieurs méthodes sont disponibles (Les getters et setters étant disponibles pour toutes les classes du projet), nous pouvons ainsi vérifier l'état du moteur, lui assigner un nouveau moteur etc...

Nous avons fait le choix de définir les méthodes consomme qui s'occupe de soustraire la consommation de carburant du moteur de chaque réservoir comme étant une fonction amie de la classe réservoir et moteur, de même pour l'ouverture des vannes VT[n], nous avons surchargé l'opérateur plus pour que celui-ci prenne en paramètre deux réservoirs et fasse une moyenne de leurs capacités respectives.

Notez que le réservoir 2 étant plus petit celui-ci ne recevra qu'un cinquième de la capacité totale maximale.

#### 3.1.2 Pompes

A chaque réservoir sont assignés deux pompes, l'une principale et l'autre secondaire (de secours), le type de pompes est caractérisé par une énumération de type (Principale/Secondaire), la classe contient des attributs permettant de garder une trace du moteur et du réservoir auxquels elle est rattachée, ainsi que diverses fonctions pour modifier son état au fur et à mesure de la progression de la simulation

#### 3.1.3 Moteurs

Le moteur quand à lui est représenté par son numéro, la pompe et donc le réservoir qui l'alimente, en plus des getters et setters sur ces différents attributs nous retrouvant la méthode amie consomme décrite plus haut.

L'énumération représentant son état est la même que celle choisie pour représenter l'état des pompes

#### **3.1.4 Vannes**

Les vannes étant de deux types différents nous avons décidé de les représenter par une hiérarchie de classes où tout les comportements communs sont définis dans la classe Valve.h.

Les sous classes en ont en commun l'attribut nom, ainsi que l'attribut état (Enum : Ouvert ou Ferme)

#### **3.1.5 Vannes Pompes/Moteurs**

Les vannes reliant les pompes et les moteurs permettent grâce à la méthode alimente d'établir le lien, l'ouverture de ces vannes fait appel à la fonction alimente

#### **3.1.6 Vannes Reservoir/Reservoir**

Les vannes s'occupant de vérifier la possibilité de transfert de carburants entre les réservoirs, fait appel à l'opérateur surchargé +.

#### **3.1.7 Systeme**

Cette classe contient un tableau de 3 réservoirs, de 3 moteurs et de vannes qui sont initialisées à la construction de l'objet, cet objet systeme sera utilisé tout au long de la simulation avec les paramètres initiaux qui lui ont été attribués .

Le systeme peut mettre à jour sa capacité globale ainsi que la capacité de tout ces réservoir grâce aux méthodes Systeme::UpdateCapaciteMax() et Systeme::UpdateConso(), la fonction amie friend void apply() permet d'appliquer une à une les pannes de la structure retournée par le parser sur le systeme lui même

### **3.2 States**

Pour faciliter les transitions et l'isolation entre les interfaces graphiques et leurs logique, nous avons mis au point un système d'états et de pile (StateManager), par exemple: Au début du programme, l'état de connexion (LoginState) est ajouté au sommet de la pile, il devient l'état courant, une fois le pilote connecté, l'état suivant est ajouté au sommet de la pile, revenir en arrière revient à dépiler un état et revenir à l'état précédent.

Chaque état appelle ainsi la fenêtre graphique correspondante et applique un traitement qui lui est propre, ajouter une nouvelle interface graphique revient à ajouter l'état et l'interface correspondante

### 3.2.1 State

Tous les états du système sont dérivés de la super classe abstraite State.h, les comportements communs à toutes les états y sont regroupés.

Les méthodes communes à tous les états sont:

- **init:** A la construction de l'état, cette fonction est appelée pour permettre d'initialiser l'état courant (Allocation mémoire, initialisation de variables etc..)
- **free:** A la destruction de l'objet s'occupe de la libération de la mémoire, la fermeture des fichiers ouverts et la fermeture de la connexion à la base de données
- **display:** s'occupe de l'affichage d'une "Frame" de l'interface graphique correspondant à l'état.
- **update:** S'occupe de mettre à jour l'interface graphique suite à une mise à jour de l'état et du système

### 3.2.2 StateManager

Cette classe principale déclarée et initialisée au début du programme, s'occupe de gérer les états courants, on peut récupérer le sommet de la pile contenant un état et y appliquer les méthodes décrites en 3.2.1.

### 3.2.3 LoginState

Login State est le premier état à être empilé sur la pile, après avoir appelé l'interface graphique correspondante, cet état ouvre dans la méthode `init()` une connexion à la base de données SQLite et exécute la Query suivante :

```
SELECT * FROM Pilotes WHERE login = 'login '
```

La méthode `isUser` renvoie vrai si les informations correspondent avec les données saisies par le pilote et faux sinon, cette variable booléenne sera utilisée plus tard dans le GUI correspondant pour déclencher l'empilement du prochain état.

### 3.2.4 MenuState

Cet état est empilé directement après la validation des informations de connexion par le pilote, la logique de cet état n'est pas très complexe, l'essentiel du travail se fait côté GUI.

Une fois le mode de simulation choisi par l'utilisateur, le prochain état empilé est le `MainGState`.

### 3.2.5 MainGState

C'est l'état principal de la simulation il peut être construit selon deux méthodes (Constructeurs), l'un prenant des paramètres définis dans lors de la simulation libre et lors depuis le fichier de configuration XML.

Une fois la simulation terminée, le score est affiché au pilote et celui ci est renvoyé à l'écran de menu.

## 3.3 GUI

Toutes nos interfaces graphiques héritent de la classe QWidget fournie dans Qt cela permet de combiner les widgets entre eux et d'utiliser la fonctionnalité propre à Qt de signaux et de slots

### 3.3.1 LoginGui

Interface graphique correspondant à l'état LoginState, composé d'une icône QIcon et de deux champs textes(QLineEdit), ainsi que de deux boutons (QPushButton) : connect et quit, qui respectivement à l'appel de leurs signaux clicked() déclenchent les méthodes checkcred qui vérifie si c'est un utilisateur est dans la base, passe au prochain état si oui sinon affiche un QMessageBox::Warning informant l'utilisateur qu'il y'a eu une erreur

## 3.4 MenuGui

Interface graphique correspondant à l'état MenuState, composé de 4 boutons (QPushButton)

- Nouvelle Simulation et Charger Simulation: Ouvrent une boîte de dialogue (QFileDialog et sa méthode statique QFileDialog::getOpenFileName()) demandant au pilote de choisir un fichier, une fois les vérifications d'extensions faites et le fichier parser par le parser, la simulation commence ou reprend
- Simulation Libre: Ce bouton une fois enfoncé affiche une boîte de dialogue ( QInputDialog::getText() qui récupère un entier entre 0 et 2 permettant de lancer une simulation aléatoire selon la difficulté choisie par le pilote
- Quitter: Quitte l'application en appelant le signal Quit() de QApplication qui est l'application principale initialisée dans la fonction main.

### 3.4.1 MainGui

## 3.5 Utils

### 3.5.1 Fichiers de configuration

Pour les fichiers de sauvegarde et de configuration de la simulation, nous avons opté pour le format de fichiers xml, car le framework QT intègre un module

complet et bien documenté sur le parsing de fichiers xml.

Le format du fichier de configuration se fait comme suit:

```
<Simulateur duree="15" tempsactuel="5" capacite ="1000" consommation = "2.6" >
  <Pannes>
    <Panne id="1" idparent="1">
      <Piece>1</Piece>
      <nbPiece>2</nbPiece>
      <Note>0</Note>
      <Temps>5</Temps>
      <Passe>0</Passe>
    </Panne>
    <Panne id="2" idparent="-1">
      <Piece>0</Piece>
      <nbPiece>1</nbPiece>
      <Note>0</Note>
      <Temps>8</Temps>
      <Passe>0</Passe>
    </Panne>
    <Panne id="3" idparent="-1">
      <Piece>0</Piece>
      <nbPiece>1</nbPiece>
      <Note>0</Note>
      <Temps>8</Temps>
      <Passe>0</Passe>
    </Panne>
  </Pannes>
</Simulateur>
```

La balise Simulation ainsi que ces attributs contient les paramètre de la base d'une simulation, la durée totale, le temps où dernière sauvegarde à été effectué, la capacité maximum du réservoir, ainsi que la consommation de base des moteurs.

Ces données seront parsés par le xmlparser pour construire un objet de type Systeme.

La balise englobante pannes contient tous les pannes par ordre d'arrivée dans la simulation, on y retrouve des balise pannes contenant les informations relatives à chaque pannes (cf Section Pannes).

### 3.5.2 Parser

Xmlparser lit dans un fichier de type QFile les différentes balises et s'occupe d'initialiser le système ainsi qu'une structure de donnée (QVector similaire à std::Vector) servant à faciliter le traitement et l'application des pannes au système.

Dans le cas d'une simulation libre, le parser, en appelant la méthode sta-



tique `xmlparser::GetRandomPannes()` renvoie un tableau de pannes selon les paramètres dépendant de la difficulté de la simulation choisie par le pilote

### 3.5.3 Pannes

Les pannes pouvant être construites de deux façons différentes selon que la simulation soit libre ou non, contient les attributs suivant que l'on retrouve dans le fichier xml.

L'id de la panne représentant son ordre de passage dans le système, le numéro du parent dans le cas d'une pompe pour garder la trace du réservoir associé, dans les autres cas -1, le type de panne (Pompe ou réservoir) est représenté par un entier valant respectivement 0 et 1, le numéro de la pièce du système est lui aussi représenté par un entier de 1 à 3, nous avons ensuite la note attribuée au pilote pour la gestion de cette panne, le temps en seconde depuis le début de la simulation représentant le moment où la panne est appliquée au système par la fonction `apply`.

L'injection des pannes dans le système est représenté par la modification des états des pièces correspondantes qui se répercute sur l'affichage de la fenêtre principale par un changement de couleur.

Dans le cadre de la simulation libre, une fonction privée de la classe permet la génération de nombres aléatoires selon la valeur maximale passée en paramètre. La variable `passé` définit si la panne a déjà été appliquée au système dans le cadre d'une reprise de partie sauvegardée.

### 3.5.4 Notation

### 3.5.5 Sauvegarde

Tout comme le parsing du fichier xml, les fichiers de sauvegarde, pouvant être générés à n'importe quel moment de la simulation, respectent le même format que les fichiers de configuration.

La classe `xmlparser` récupère dans les paramètres de la méthode statique `void WriteintoXmlFile()`, les informations concernant le système, les pannes et les notes au moment de la sauvegarde.

Le fichier ainsi créé est nommé avec le pseudo du pilote concaténé avec le timestamp courant.

## 4 Ressources

Le répertoire `assets` contient tout les fichiers hors code source permettant le bon fonctionnement du simulateur.

Le sous répertoire `pannes` contient les fichiers XML des simulations de test fournies avec le Simulateur.

On y retrouve aussi les différents logos et icônes du projet.

## **5 Listing**

Est joint en annexe de ce document une documentation réalisée à partir de doxygen, consultable dans le répertoire du projet dans le sous dossier docs/.