|  |
| --- |
| HES-SO |
| Projet C++ : Logic Emulator |
| Rapport |

|  |
| --- |
| Gilles Mottiez  17/06/2018 |

Table des matières

[Introduction 2](#_Toc517005332)

[But et cahier des charges 2](#_Toc517005333)

[Développement 2](#_Toc517005334)

[Conception globale 2](#_Toc517005335)

[Packages 2](#_Toc517005336)

[Patterns 3](#_Toc517005337)

[Chargement des fichiers 3](#_Toc517005338)

[Décodage du fichier Json 4](#_Toc517005339)

[Génération du modèle logique 4](#_Toc517005340)

[Calcul des sorties selon le type de portes et des entrés 5](#_Toc517005341)

[Affichage graphique 5](#_Toc517005342)

[Modification des entrées et validation des modifications 5](#_Toc517005343)

[Problèmes et solutions 5](#_Toc517005344)

[Résultats 6](#_Toc517005345)

[V0.0 6](#_Toc517005346)

[V1.0 6](#_Toc517005347)

[V2.0 6](#_Toc517005348)

[Conclusion 6](#_Toc517005349)

# Introduction

Afin de mettre en pratique les connaissances acquises lors du 3ème et 4ème semestre du cours Inf2, nous avons dû créer le software de notre choix. Ce software se devait de répondre à certaines exigences, définies dans le cahier des charges. De plus, il devait être conçu selon des patterns et des règles fiables, étudiées lors de l’année.

Mon choix s’est porté sur un émulateur de logique numérique. J’ai choisi ce sujet car je le trouve concret et il permet d’associer 2 branches enseignées aux cours.

# But et cahier des charges

Concevoir un software qui permette la simulation d’un système de logique numérique. Le fonctionnement est le suivant :

* Des fichiers JSON sont écrit selon un modèle précis, qui est défini dans l’annexe « JSON\_logicProtocol ».
* Le software permet de charger un de ces fichiers, de générer un modèle de ce fichier en porte logiques, d’en calculer l’état de ses portes et d’afficher une représentation graphique.
* Grâce à une fenêtre textuelle, l’utilisateur peut changer des valeurs d’entrées sans avoir à modifier tout le fichier et à le recharger complétement

Versions :

* V0.0
  + Charger un fichier Json dans le software
  + Implémentation des portes AND, OR, NOT
  + Création de la représentation graphique
  + Génération de la logique
* V1.0
  + Création de bloc logiques tels que XOR, NAND, NOR, …
  + Possibilité d’éditer le code directement à l’écran pour la modification des signaux d’entrée
* V2.0
  + Bascule D

# Développement

## Conception globale

Ce software a été conçu selon la théorie du cours « software engineering ». Divers patterns ont été utilisés et sont décrit ci-dessous.

### Packages

* UserInterface : ce package contient la classe IOView qui s’occupe de l’affichage des données à l’écran, ainsi que la récupération des informations saisies par l’utilisateur
* Controller : centre névralgique du software, il contient la classe du même nom, qui est la machine d’état du système. Elle s’occupe de traiter les événements et de donner la voie à suivre.
* Data : il s’agit du modèle du système, c’est ce package qui contient toutes les données, ainsi que les méthodes qui permettent de les traiter, sur les ordres du Controller.
* Interfaces : regroupe toutes les interfaces de communication entre les différents packages grâce au pattern des « Port » de communication.

### Patterns

#### MVC :

Pour Model View Controller. Il s’agit d’un découpage du logiciel en fonctions primaires telles que la récolte de données, la prise de décision quant à la route é suivre, le traitement des données et enfin l’affichage des résultats.

#### Machine d’états et XFEvents :

Elle permet de gérer de manière centralisée le déroulement du processus. Elle a l’avantage de contenir toute la gestion événementielle du logiciel en un seul endroit, ce qui garantit la robustesse. Nous utilisons un XF, qui gère de manière transparente pour le programmateur le traitement des erreurs. Il ne nous reste qu’à envoyer des XFEvents au bon moment, et de les traiter ensuite convenablement.

#### Double switch pattern :

Implémenté dans la machine d’état de la classe Controller, il permet de gérer les passages d’un état à l’autre. Il propose d’utiliser des méthodes « on transition, on entry, on exit ». Pour cela, on utilise les XFEvents qui permettent de switcher entre les états.

#### Portes de communications :

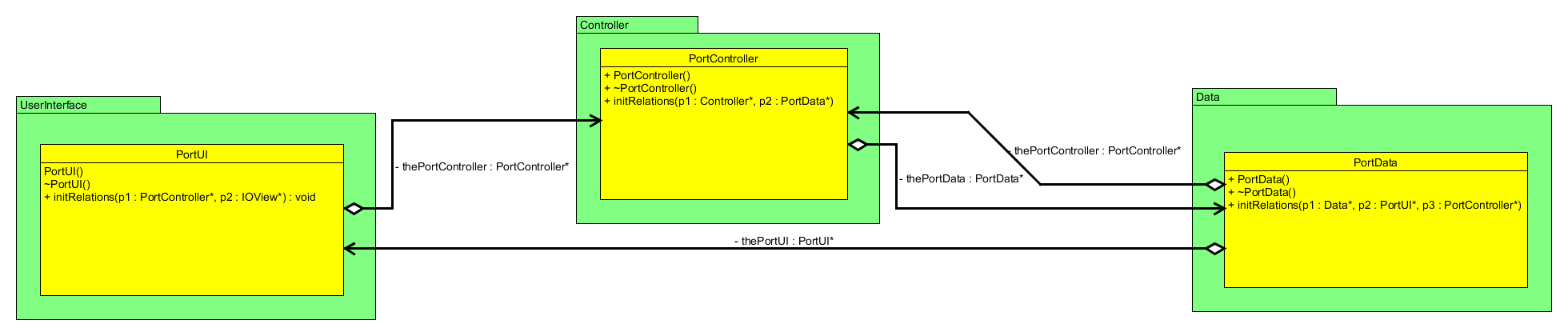
Elles permettent l’échange de données entre les différents packages. Chaque package à sa propre porte qui permet de communiquer avec les autres portes. Les méthodes sont strictement définies dans les Interfaces.

Figure 1 : portes de communication

Ce diagramme permet aussi de comprendre le MVC : une notification parvient au Controller depuis la UserInterface. Le Controller décide de quoi faire, il notifie à son tour le Model, qui va mettre à jour ses données, puis le Model renvoie au Controller ses nouvelles informations afin qu’il prenne une décision. Finalement le Controller communique au Model la direction à prendre, soit le nouvel état, et le Model s’occupe d’envoyer les données à la UserInterface pour afficher les résultats à l’écran.

## Chargement des fichiers

Lorsque l’utilisateur clique sur le bouton « load », un explorateur de fichier s’ouvre et l’utilisateur peut choisir un fichier Json précédemment écrit à charger. La classe QFileDialog est utilisée. Elle permet de gérer ceci très facilement et de récupérer un QString contenant le chemin du fichier voulu.

Un QFile\* est ensuite créé grâce au chemin du fichier, il suffit de lire ce QFile\* avec un QTextStream et sa méthode readAll() et le tour est joué.

QString IOView::getPath()

{

//set the filter to get only .json files

QString filter = "File Description (\*.json)";

//get the opened file's path

QString filePath = QFileDialog::getOpenFileName(

this,

"Open Json file",

QDir::homePath(), filter);

return filePath;

}

## Décodage du fichier Json

Qt fournit 3 classes hautement utiles à la lecture d’un fichier Json. Elles se nomment :

* QJsonDocument : création d’un objet à partir d’un fichier Json
* QJsonObject : création d’un objet à partir d’un élément d’un fichier Json
* QJsonArray : création d’un tableau soft à partir d’un tableau Json

Il est ainsi relativement aisé de décomposer un fichier Json en suivant une certaine méthodologie, qui consiste à procéder de l’extérieur vers l’intérieur :

//get the name of the file

QString fileName = design["name"].toString();

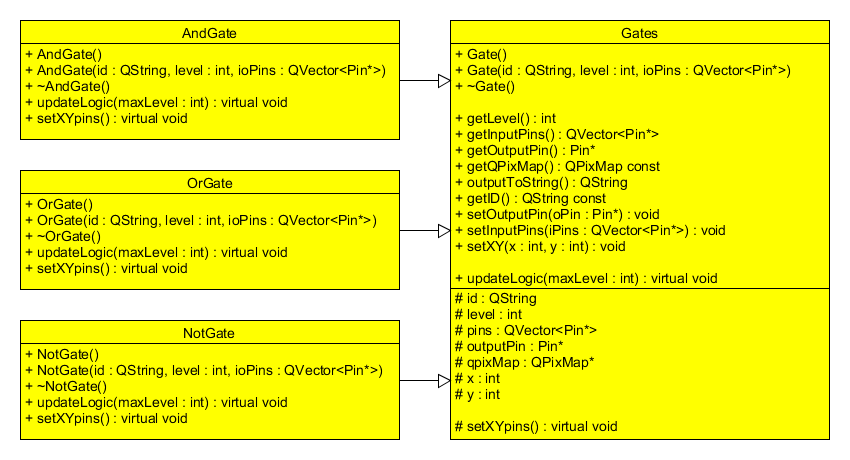
//get the array of all the gates

QJsonArray gates = design["gates"].toArray();

* Récupération d’un objet QJsonDocument à partir d’un QByteArray obtenu par le QString
* Récupération d’un objet QJsonObject à partir de l’objet QJsonDocument
* Récupération des attributs en utilisant l’opérateur [] d’un QJsonObject et en spécifiant le nom de la variable à récupérer ainsi que le type de variable dans lequel on veut stocker sa valeur

## Génération du modèle logique

À partir des variables récupérées du fichier Json, il faut créer des objets héritant de la classe Gate, de type AndGate, OrGate ou encore NotGate.



//create a logic gate

if(id.contains("AND"))

{

AndGate\* andGate = new AndGate(id, level, vPinsIO);

vGates.push\_back(andGate); //add to the global vector

}

else if(id.contains("OR"))

{

OrGate\* orGate = new OrGate(id, level, vPinsIO);

vGates.push\_back(orGate); //add to the global vector

}

else if(id.contains("NOT"))

{

NotGate\* notGate = new NotGate(id, level, vPinsIO);

vGates.push\_back(notGate); //add to the global vector

}

Figure 2 : héritage de la classe Gate

À chaque fin de récupération des attributs correspondant à une Gate, un pointeur est créé selon le type de Gate et stocké dans un QVector<Gate\*> vGates, de la classe Data. La puissance du polymorphisme est ainsi utilisée.

## Calcul des sorties selon le type de portes et l’état des entrées

La prochaine étape est de calculer les états de chaque Gate. Pour cela, on traite d’abord les Gates avec la variable level == 0. Ce sont elles qui ont les signaux d’entrées, soit « LOG\_HIGH » soit « LOG\_LOW ». Chaque Pin d’entrée voit ainsi son booléen « state » mis à jour. Ensuite, la Pin de sortie est connectée à la Pin correspondante, et sa valeur est calculée selon le type de porte et l’état des entrées.

Ensuite, les Gates intermédiaires sont traitées selon le même processus, mais au lieu de lire des « LOG\_LOW » ou « LOG\_HIGH », on récupère la Pin connectée grâce à son label et à la méthode getPinFromLabel(QString labelPinToFind). L’état de cette Pin connectée est copié sur la Pin d’entrée. Puis la suite se passe de la même manière pour la Pin de sortie.

Pour la dernière Gate du circuit, la seule différence est qu’elle ne contient pas de Pin connectée.

## Affichage graphique

## Modification des entrées et validation des modifications

# Problèmes et solutions

# Résultats

## V0.0

## V1.0

## V2.0

# Conclusion