



CentraleSupélec

2^e année du cursus ingénieur Supélec - Projet long

Conception d'un logiciel de simulation de circuit électrique

Pierre BIRET et Victor FAUTH, promotion 2019

Année 2017-2018

Professeur encadrant : Amir ARZANDÉ

Table des matières

Introduction	1
1 Montée en compétences : programmation d'un programme simple	2
1.1 La dualité C#/XAML	2
1.2 Règles du jeu	3
1.3 Implémentation	3
1.3.1 Le modèle MVC	3
1.3.2 Le contrôleur	4
1.3.3 La vue	5
1.3.4 Annexe	7
2 Cahier des charges et organisation	11
3 Implémentation : manipulation et affichage du circuit (front-end)	12
3.1 La classe MainWindow	13
3.2 La classe BoardGrid	14
3.3 La classe Node	14
3.4 Les classes Component et ComponentDialog	14
3.5 Les classes Wire et WireDragger	14
3.6 La classe SavedCircuit	14
4 Implémentation : simulation (back-end)	16
5 Code source et dépendances	17
Glossary	18

Introduction

Dans le cadre de notre projet long de la deuxième année du cursus ingénieur à Supélec, nous devons développer un logiciel de simulation de circuit électrique. Il s'agit d'un logiciel qui permet à l'utilisateur de créer graphiquement un circuit électrique (à l'aide de composants génériques ou personnalisés) puis de simuler son fonctionnement dans différentes conditions. Un logiciel commercial de ce type parmi les plus connus est LTspice.

Ce projet ayant déjà été proposé l'an dernier, il a été décidé que nous commencerions à partir de zéro afin de bien maîtriser chaque étape de la conception du logiciel. Cependant, nous conservons le langage de programmation précédemment utilisé. Le logiciel sera donc codé en C#. Il s'agit d'un langage orienté objet, très inspiré du C++, mais développé par Microsoft dans le cadre de la plateforme .NET et présentant de nombreuses différences. Pour gérer le front-end, nous utiliserons WPF (*Windows Presentation Foundation*).

D'un point de vue organisationnel, le projet se déroulera en plusieurs phases : tout d'abord une phase de montée en compétences en codant un petit jeu, puis l'implémentation d'une interface graphique basique pour le programme. Cela nous permettra ensuite d'ajouter la simulation du circuit. S'il nous reste du temps, nous pourrons ajouter des fonctionnalités au programme pour le rendre plus agréable, intuitif et rapide à utiliser.

I. Montée en compétences : programmation d'un programme simple

Afin d'apprendre à coder en C# avec WPF, nous avons décidé de commencer par un projet bien plus modeste : le développement d'un jeu de Memory. Il s'agit d'apprendre à coder en C# avec WPF (et donc à utiliser XAML, comme expliqué au prochain paragraphe), mais aussi de se familiariser avec l'IDE utilisé. Nous avons choisi pour cela Microsoft Visual Studio : le langage C# a été créé pour être utilisé avec cet IDE, il permet de générer le code XAML à partir d'une interface graphique, possède des outils de débogage très puissants et une licence est fournie aux élèves de Supélec. Comme gestionnaire de versions, nous utilisons Git.



FIGURE 1.1 – Les logos de Visual Studio 2017 (gauche) et de Git (droite).

1.1 La dualité C#/XAML

Lorsqu'un programme graphique est créé en utilisant WPF, deux langages sont utilisés. Les éléments graphiques (fenêtre, boutons, images, etc...) sont codés en XAML. Il s'agit d'un langage descriptif dérivé du XML. Il nous permet de décrire les attributs de chaque élément : par exemple, la fenêtre a un nom (utilisé pour l'appeler dans le code), un titre, une taille, une position, etc... Elle possède aussi des « children » : par exemple, un bouton. Ce bouton possède des attributs similaires, mais aussi certains attributs spécifiques, tels l'animation à effectuer lors de l'activation. Si une fonction à lancer au clic (ou lors de n'importe quel événement ayant lieu sur l'élément) peut être définie pour la fenêtre, cet attribut est obligatoire pour un bouton.

Le XAML est un langage qui permet ainsi de définir rapidement une interface utilisateur, sans qu'il soit nécessaire de taper trop de code. Tous les éléments sont aussi personnalisables si nécessaire, bien que cela soit assez complexe.

Le code lui-même est écrit en C#. Il gère toute la logique et l'interactivité du programme. Si une interactivité est nécessaire, par exemple pour changer le titre de la fenêtre au clic, il est possible de modifier tous les éléments depuis le code, d'en créer ou d'en supprimer.

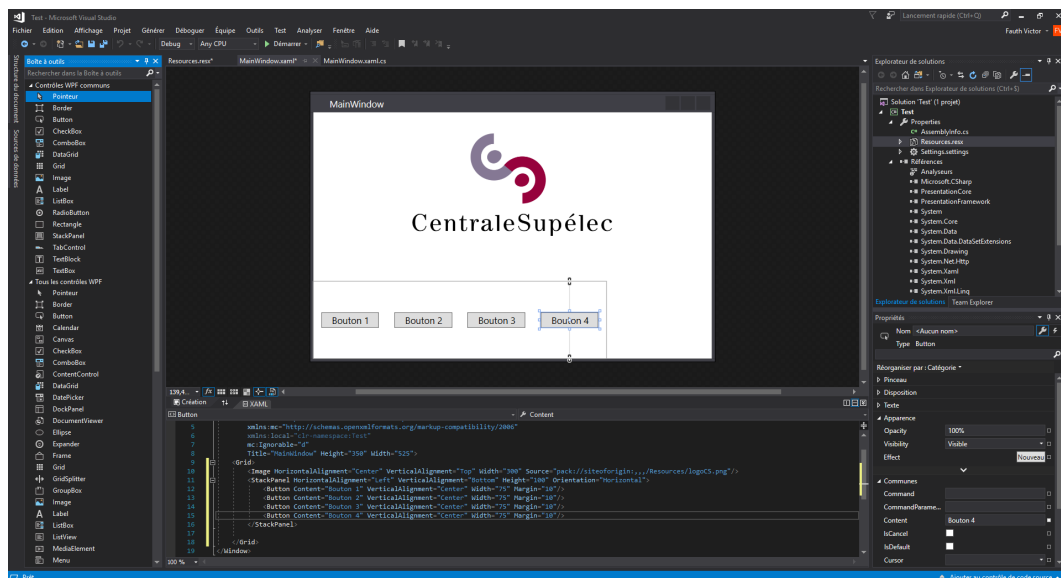


FIGURE 1.2 – L’éditeur graphique de XAML. Ici, une fenêtre avec une image et une ligne de boutons a été créée grâce à l’outil graphique, le code XAML correspondant est affiché dans la fenêtre inférieure. La fenêtre de droite permet de modifier les propriétés de l’élément sélectionné.

1.2 Règles du jeu

Il s’agit d’un jeu très simple dans lequel un certain nombre de cartes sont posées, face cachée. Ces cartes vont par paire : chaque motif est représenté sur deux cartes. Les cartes sont posées au hasard, puis le joueur retourne deux cartes. Si elles sont identiques, la paire est retirée du jeu. Sinon, elles sont retournées face cachée, au même endroit (après un temps, ici deux secondes, permettant au joueur de mémoriser les cartes). Le jeu s’arrête lorsque toutes les paires ont été trouvées, le but étant de minimiser le nombre d’essais.

1.3 Implémentation

1.3.1 Le modèle MVC

Pour coder le jeu, nous avons décidé d’utiliser l’architecture MVC (Modèle-Vue-Contrôleur). Le principe est de séparer le code en trois composantes distinctes : le modèle contient les données du programme, le contrôleur gère la logique et la vue interagit avec l’utilisateur.

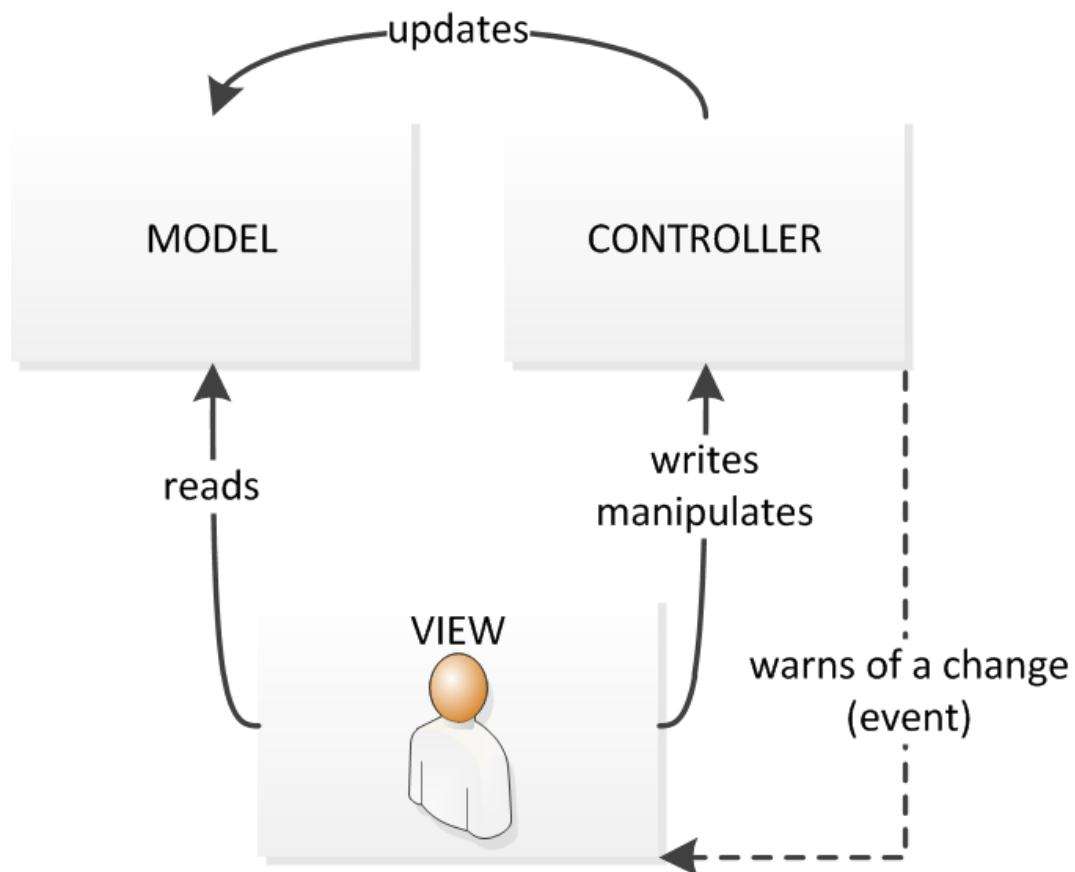


FIGURE 1.3 – Les interactions de l’architecture MVC. Image tirée de Wikipédia.

Remarque La structure de chaque classe est donnée en sous-section 1.3.4.

Le modèle

Le modèle contient l’état du jeu à un moment donné. Pour cela, nous définissons deux classes : la classe `Card` et la classe `Board`. `Card` représente une carte et définit son état (trouvée et/ou affichée) ainsi que la paire dont fait partie la carte, tandis que `Board` représente tout le plateau de jeu et contient notamment toutes les cartes, ainsi que le compteur de tours pour la partie en cours. Il possède une méthode pour incrémenter ce compteur, appelée à chaque fois que le joueur a désigné deux cartes, et deux méthodes utilisées lors de l’initialisation du plateau de jeu.

1.3.2 Le contrôleur

Le contrôleur a la responsabilité de toute la logique du jeu. Il possède peu d’attributs, le stockage des données étant du ressort du modèle : deux attributs servent juste à référencer la vue et le plateau de jeu, le troisième enregistre quelle carte a été choisie par le joueur en attendant qu’il en choisisse une seconde.

Il y a cependant bien plus de méthodes, qui sont appelées lorsque le joueur agit. La méthode `CardChosen` est appelée lorsque le joueur choisit une carte. S’il en a déjà choisi une juste avant, cette méthode vérifie si les deux cartes forment une paire avec `CheckPair`, puis agit en conséquence. La méthode `NextTurn` est alors appelée, et prépare le tour suivant, notamment en cachant à nouveau les cartes choisies précédemment (si elles ne constituaient pas une paire). C’est aussi elle qui va gérer

la fin du jeu avec les méthodes `IsFinished` (qui renvoie un booléen indiquant si toutes les paires ont été trouvées ou non) et `Exit` qui demande à la vue d'afficher la fenêtre de fin de partie.

1.3.3 La vue

L'interface graphique statique en XAML

Le jeu gère n'importe quel nombre (pair) de cartes, nous n'avons donc pas défini l'interface graphique de manière statique en XAML : la classe `Display` se charge de créer la totalité de l'affichage et de charger les images présentes dans le dossier pour les afficher en fonction du nombre de cartes souhaité. Le code XAML est donc presque vide, seule un élément `Grid` est défini. Celui-ci s'étend sur l'intégralité de la surface de la fenêtre et permet juste de positionner facilement les cartes.

L'interactivité en C#

La classe `Display` contient donc toutes les méthodes pour charger les images, créer les cartes à partir de ces images et des données du modèle, afficher et rafraîchir le plateau de jeu, et afficher la fenêtre des scores en fin de partie. Il s'agit de la classe la plus complexe à coder : pas à cause de sa logique, mais parce que cela a nécessité de bien comprendre le fonctionnement de WPF. Si le contrôleur et le modèle furent assez simple à coder, c'est parce qu'il s'agissait de programmation « générique », à laquelle nous sommes déjà habitués, et le C# n'est pas un langage extrêmement complexe à prendre en main si nous sommes déjà habitués au C++ ou au Java. Cependant, le WPF est extrêmement particulier, et, bien qu'il soit extrêmement puissant, de nombreux concepts sont difficiles à prendre en main et nécessitent de se plonger dans la documentation. Cela fut très instructif, nécessaire pour la suite du projet, et surtout formateur.

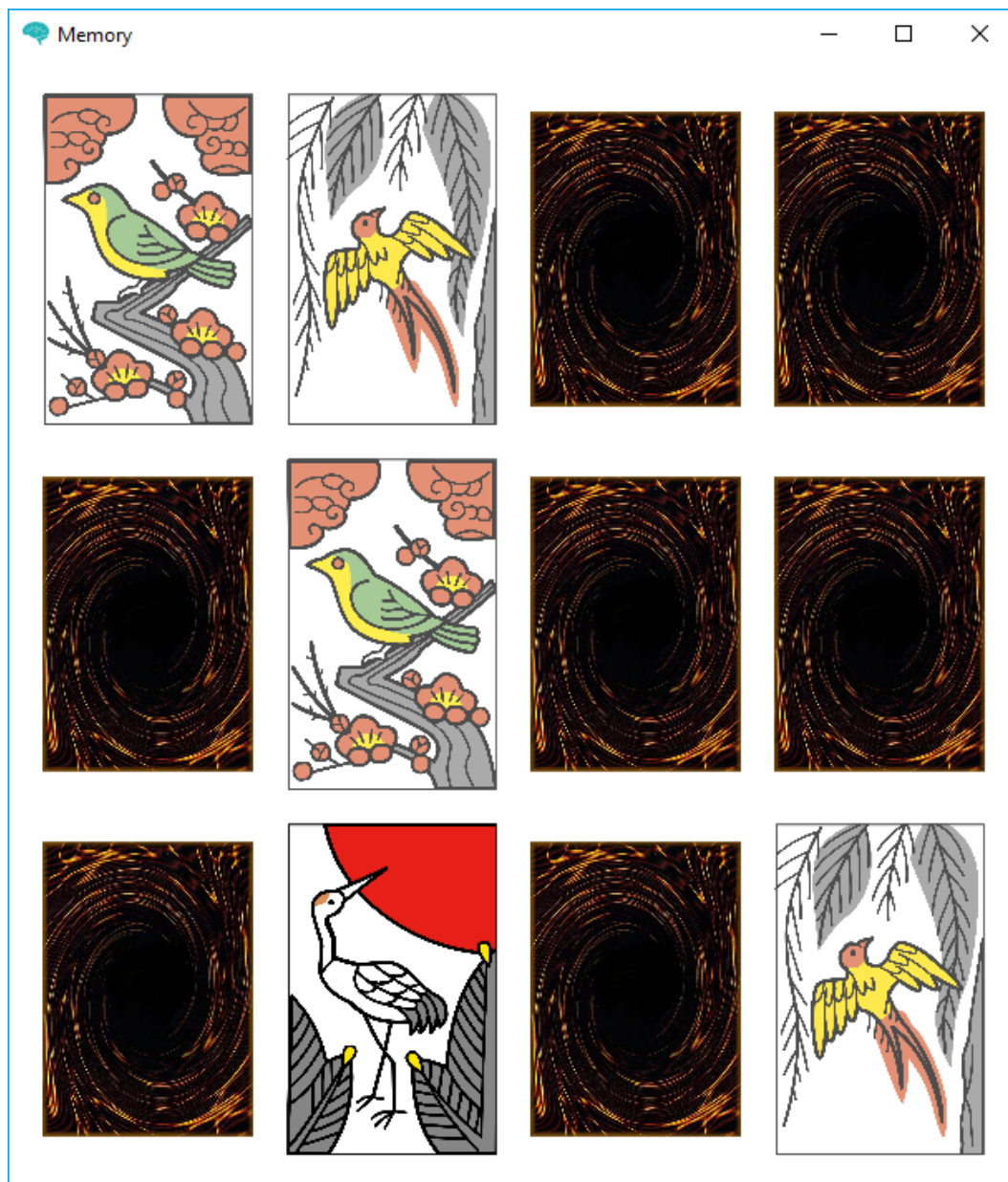


FIGURE 1.4 – Une partie avec 16 cartes. Deux paires ont été trouvées, et une carte a été sélectionnée.

1.3.4 Annexe

Listing 1.1 – La structure de la classe Card du modèle.

```
class Card
{
    // Attributes

    private Card cardPaired; // The other card in the pair
    private int pair; // The id of the pair
    private bool isDisplayed; // Whether the card is currently
        displayed
    private bool isFound; // Whether this card's pair has been
        found

    // Accessors/Mutators

    /*
    [...]
    */

    // Constructor

    /// Create one card
    public Card(int pair)
}
```

Listing 1.2 – La structure de la classe Board du modèle.

```
class Board
{
    // Attributes

    private Card[][] cards; // Array of cards sorted by their
        position
    private int turns; // The number of turns played
    private int X; // The number of colons of cards
    private int Y; // The number of lines of cards

    // Accessors/Mutators

    /*
    [...]
    */

    // Constructor

    /// Create the board, the pairs of cards, and randomize their
        positions
    public Board(int X, int Y)

    // Methods

    /// Add one turn to the counter
    public void IncrementTurns()

    /// Shuffle an array of any objects
    public static void ShuffleArray(object[] array)

    /// Check whether each paired cards have the same symbols. If
        not, correct that
    private void CheckSymbols()
}
```

Listing 1.3 – La structure de la classe Game du contrôleur.

```
class Game
{
    // Attributes

    private Card firstCardSelected; // The first selected card
        this turn
    private Board board; // The board with all the cards
    private Display display; // The GUI

    // Accessors/Mutators

    public Board GetBoard()

    // Constructor

    /// Create a new game
    public Game(Display display, int X, int Y)

    // Methods

    /// Method to be executed when a card is chosen
    public void CardChosen(object sender, EventArgs e)

    /// Go to the next turn by incrementing the counter and
        refreshing the display
    public void NextTurn()

    /// Check whether two cards are paired
    private bool CheckPair(Card card1, Card card2)

    /// Check whether the game is finished or not
    private bool IsFinished()

    /// Exit the game
    public void Exit(object sender)
}
```

Listing 1.4 – La structure de la classe Display de la vue.

```
class Display
{
    // Attributes
    private MainWindow mainWindow; // The window
    private Game game; // The controller
    private Board board; // The board
    private Card[][] cards; // The cards
    private Image[][] images; // The objects used to display the
        cards in WPF

    private BitmapImage[] symbols; // Pictures to show on the
        cards
    BitmapImage hidden; // Picture to show on hidden cards

    public const int CARD_X = 127; // Width (in pixels) of every
        card
    public const int CARD_Y = 200; // Height (in pixels) of every
        card

    // Constructor

    /// Create a graphical interface for the game and initialize
        the game itself
    public Display(MainWindow mainWindow, int X, int Y)

    // Methods

    /// Load all the images present in the ./resources/cards
        directory
    public void LoadImages(int number)

    /// Create the graphical controls for the board and the cards
    public void CreateBoard()

    /// Update the display
    public void PrintBoard()

    /// Create an image
    public Image CreateCard(Panel panel, int x, int y, int width,
        int height, int marginLeft = 0, int marginTop = 0, int
        marginRight = 0, int marginBottom = 0)

    /// Display a pop-up with the number of turns
    public void PrintTurns()
}
```

II. Cahier des charges et organisation

Le but est de développer un logiciel permettant la saisie de circuits électriques ainsi que la simulation de leur comportement. Il n'est pas attendu un logiciel de qualité professionnelle devant être effectivement utilisé : nous nous limitons donc à des fonctionnalités d'édition basiques (création, sauvegarde et chargement de circuits de taille limitée), avec une liste de composants disponibles réduite (source de tension alternative ou continue, source de courant, résistance, inductance, condensateur, fil et terre) et un fonctionnement de la simulation limité à des cas simples. Il n'y a pas de liste spécifique de capacités à remplir, mais nous avons une année pour faire du mieux possible.

Le professeur encadrant nous ayant laissé une grande latitude organisationnelle, nous avons décidé d'organiser ce projet en deux grandes phases : tout d'abord, nous allons nous occuper du front-end, avec les capacités d'édition du circuit, puis du back-end, pour permettre les simulations. Cette première partie nous a pris 2 séquences scolaires sur 4 (le Memory en avait déjà occupé une), de début décembre à début mars. La dernière séquence sera donc allouée à la simulation, avec une approche itérative : l'algorithme sera complété au fur et à mesure afin de permettre de simuler des cas plus complexes avec des résultats plus précis.

III. Implémentation : manipulation et affichage du circuit (front-end)

Afin de représenter un circuit, nous avons défini les classes suivantes : `BoardGrid` représente le « plateau » sur lequel est placé le circuit, `Node` représente les nuds de la grille du plateau, auxquels on peut connecter des éléments, `Component` représente un composant du circuit (résistance, source de courant, etc.), `Wire` représente un fil reliant deux nuds. Enfin, la classe contenant la fonction `main()` est `MainWindow`. Il s'agit de la classe représentant la fenêtre et gérant les entrées claviers et les boutons. Son code est divisé en deux parties : une partie en XAML décrit la partie statique du code (division de l'interface, boutons), une autre en C# gère les événements et l'initialisation du circuit.

Afin de gérer certaines interactions, trois autres classes sont nécessaires. Tout d'abord, déplacer un composant est facile, mais déplacer un fil, qui est composé de trois rectangles (il n'est pas forcément rectiligne), a nécessité l'ajout d'une classe `WireDragger` qui gère le dragging d'un fil. Ensuite, pour éditer les propriétés d'un composant, telle la valeur d'une résistance par exemple, il est nécessaire de créer un pop-up avec les champs nécessaires. Cela ajoute la classe `ComponentDialog`. Finalement, pour des raisons techniques qui seront expliquées plus bas, la sauvegarde du circuit nécessite l'ajout d'une classe `SavedCircuit`.

Cela donne en UML le diagramme de classes suivant (les accesseurs ne sont pas affichés pour simplifier le diagramme) :

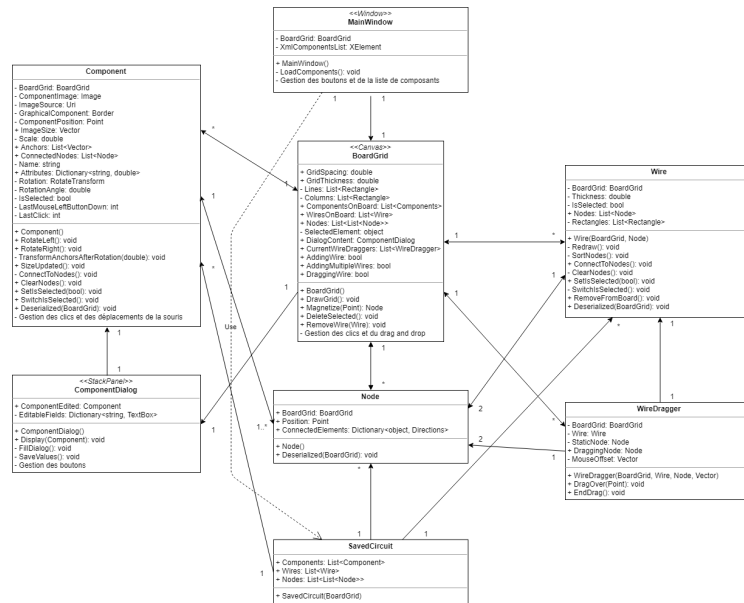


FIGURE 3.1 – Le diagramme de classes correspondant au front-end.

3.1 La classe MainWindow

Cette classe la fenêtre et les éléments qui en dépendent directement. En XAML, certains éléments sont définis de manière statique : la barre d'outils et ses boutons d'édition : nouveau circuit, chargement, sauvegarde, rotation à gauche, rotation à droite, suppression, mode d'ajout de fil et mode d'ajout de multiples fils (celui-ci n'est visible que lorsque le mode précédent est actif ; le mode simple d'ajout de fil se termine après l'ajout du premier fil, celui-ci non). Ensuite, un panneau est défini sur la gauche de l'écran pour proposer les composants pouvant être ajoutés. Sur la droite, le reste de l'espace est réservé au circuit proprement dit : lors de l'initialisation, le programme lance une instance de BoardGrid et l'affiche dans cet espace.

Cette classe contient aussi le code permettant de charger la liste des composants disponibles, qui est présente dans un fichier XML. Cela permet de modifier leurs caractéristiques rapidement et sans toucher au code source. Par exemple, la résistance est définie dans les lignes suivantes :

Listing 3.1 – Le XML définissant une source de tension alternative. Les ancres correspondent aux points de l'image sur lesquels un câble peut se brancher.

```

<ac_voltage_source>
  <type>ac_voltage_source</type>
  <image>ac_voltage_source.png</image>
  <name>Source de tension alternative</name>
  <width>922</width>
  <height>922</height>
  <anchors>
    <anchor>
      <posX>0</posX>
      <posY>460</posY>
    </anchor>
    <anchor>
      <posX>921</posX>
      <posY>460</posY>
    </anchor>
  </anchors>
</ac_voltage_source>
  
```

```

        </anchor>
    </anchors>
    <attributes>
        <attribute>
            <name>Amplitude</name>
        </attribute>
        <attribute>
            <name>Fréquence</name>
        </attribute>
    </attributes>
</ac_voltage_source>

```

3.2 La classe BoardGrid

Cette classe représente le circuit dans son intégralité. Lors de son initialisation, elle dessine une grille constituée de rectangles horizontaux et verticaux et place des `Node` aux intersections. Ensuite, elle gère l'ajout de composants et de fils, et en garde une liste. Une instance de `BoardGrid` décrit donc parfaitement un circuit.

3.3 La classe Node

La fonction `Magnetize()` de `BoardGrid` s'assure que tous les éléments (composants ou fils) sont bien ancrés sur un `Node` du circuit : les positions possibles sont discrétisées. Un `Node` conserve donc une liste de tous les éléments qui lui sont connectés. Lors de la simulation, un `Node` correspond donc à un nud du graphe représentant le circuit, et avoir une liste de toutes les instances de cette classe suffit à réaliser une simulation.

3.4 Les classes Component et ComponentDialog

Un composant doit à la fois gérer les connections de ses ancrs aux nuds du circuit, pouvoir être déplacé, subir des rotations, être sélectionné, et pouvoir éditer ses attributs. `Component` gère donc beaucoup d'événements, et gère tous les calculs pour que graphiquement, l'image représentant le composant soit toujours bien située et orientée. Lors d'un double-clic, une instance de `ComponentDialog` est créée, qui permet de modifier les attributs du composant via un pop-up dont le contenu est généré dynamiquement en fonction du composant.

3.5 Les classes Wire et WireDragger

Les fils sont des composants qui relient deux nuds. Ils ne peuvent pas être déplacés individuellement, mais si un composant connecté à un film est déplacé, le fil le suivra. Afin de gérer ces déplacements, à chaque fois une instance de la classe `WireDragger` est créée. Cette classe permet de gérer le déplacement du fil, lequel nécessite de définir est composé d'un bout fixe et d'un bout mobile qui suit la souris et se connecte à chaque mouvement au nud le plus proche du pointeur.

3.6 La classe SavedCircuit

Afin de sauvegarder un circuit, nous utilisons le principe de la sérialisation, qui consiste à encoder une classe et tous ses attributs sous forme d'informations atomiques faciles à sauvegarder et à restaurer. C# propose des outils permettant de réaliser cela très facilement. Il aurait été ainsi suffisant de sauvegarder l'instance de `BoardGrid`, qui contient en attribut des pointeurs vers toutes les autres

instances représentant le circuit, si cela avait été possible. En effet, un certain nombre des classes que nous utilisons héritent de classes définies par WPF qui, pour des raisons techniques, ne permettent pas la sérialisation.

Nous avons donc implémenté la classe `SavedCircuit`, dont une instance est créée lorsque l'on désire sauvegarder le circuit. Cette instance va lister tous les composants, fils et nuds du circuit : ces informations suffisent à le décrire parfaitement. Ensuite, l'instance de `SavedCircuit` est sérialisée. Pour charger un circuit, il suffit de la désérialiser, puis de créer une nouvelle instance de `BoardGrid` en lui donnant l'ensemble des données sauvegardées.

IV. Implémentation : simulation (back-end)

V. Code source et dépendances

Le code source complet et à jour est disponible sur les dépôts Git suivants :

- Pour le projet principal : <https://github.com/Fauth/PPcurry>
- Pour ce rapport : <https://github.com/Fauth/PPcurry-report>
- Pour le mini-projet d'entraînement : <https://github.com/Fauth/memory>

Pour pouvoir compiler le code, les dépendances suivantes sont requises :

- **ALGLIB**, une bibliothèque d'analyse numérique et de traitement de données que nous utilisons pour manipuler des matrices : <http://www.alglib.net/>
- **Material Design In XAML Toolkit**, une bibliothèque qui nous permet de faire du *flat design* afin de rendre le logiciel plus beau, et qui ajoute plusieurs contrôles utiles : <https://github.com/Fauth/PPcurry-report>

Glossaire

Back-end Le back-end représente la partie d'un logiciel (et donc le code associé) qui gère la logique et le fonctionnement interne d'un logiciel. Dans notre cas, il s'agit du code réalisant la simulation.

Front-end Le front-end représente la partie d'un logiciel (et donc le code associé) qui interagit avec l'utilisateur. Il s'agit donc de l'interface entre le back-end et l'utilisateur. Dans notre cas, il s'agit de l'interface graphique.

IDE L'environnement de développement intégré (IDE en anglais) est un logiciel permettant à la fois d'éditer du code et de le compiler. Il comprend généralement des outils simplifiant le développement, tels une coloration syntaxique, une auto-complétion du code ou un débogueur.