CentraleSupélec

2e 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

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.

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`

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 déboguage 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.

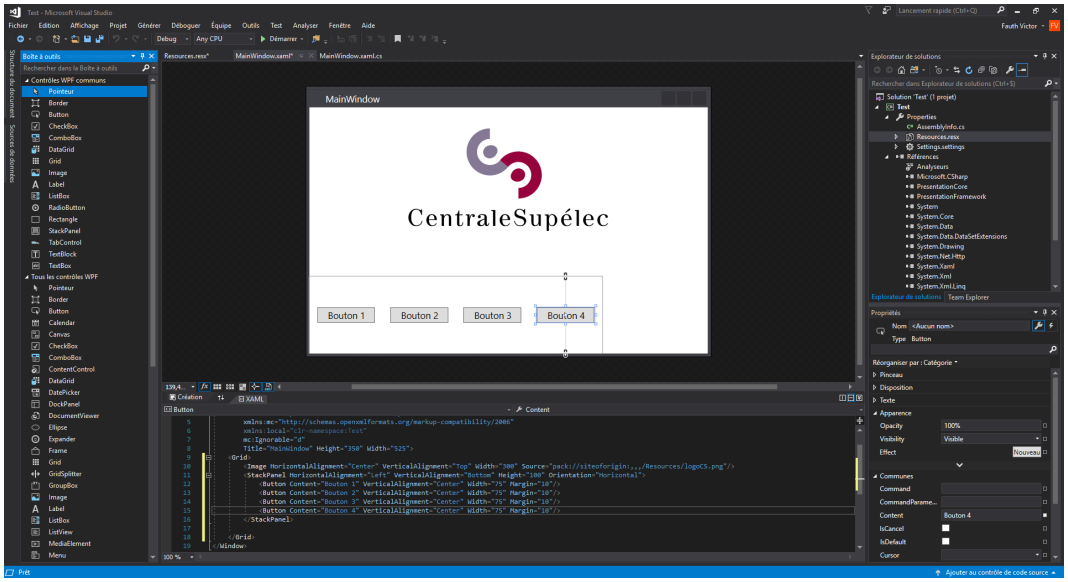

FIGURE 1.2 – L'éditeur graphique de XAML. Ici, une fenêtre avec une image et une ligne de boutons a été créée grâce à l'outil graphique, le code XAML correspondant est affiché dans la fenêtre inférieure. La fenêtre de droite permet de modifier les propriétés de l'élément sélectionné.

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.

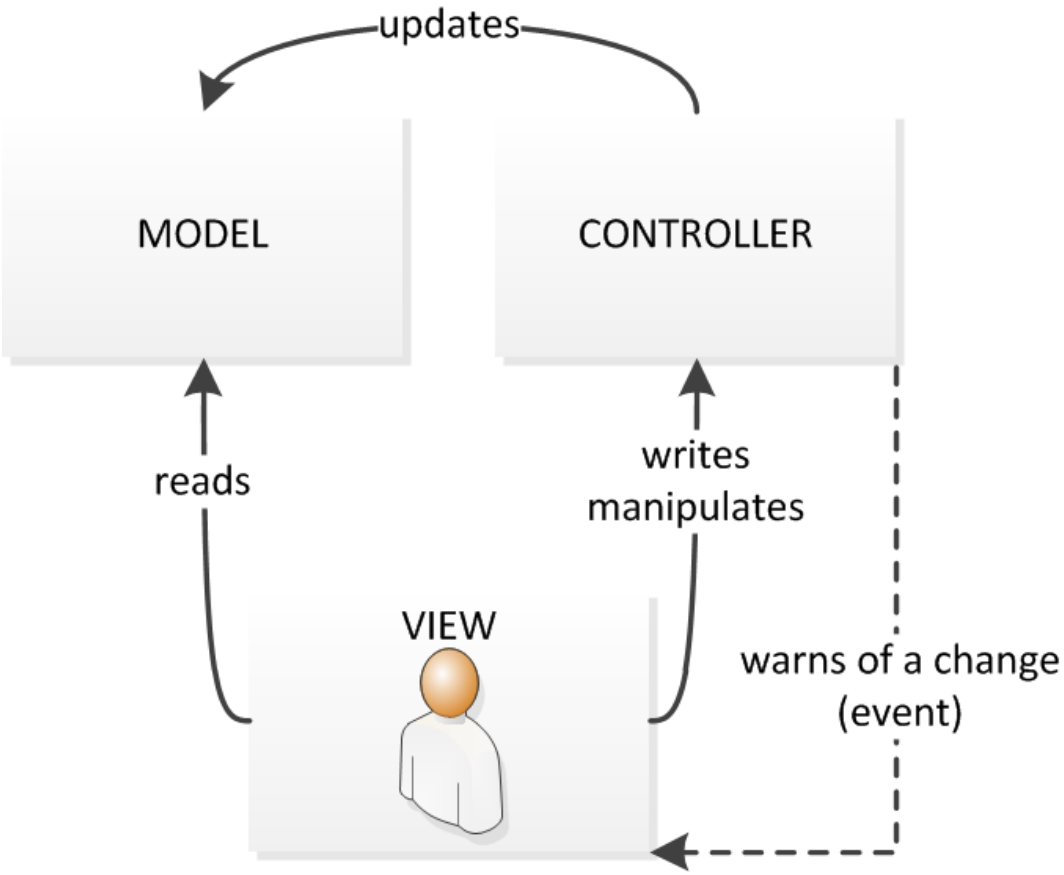

FIGURE 1.3 – Les interactions de l'architecture MVC. Image tirée de Wikipédia.

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

Le modèle

Le modèle contient l'état du jeu a 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 pair 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 une 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 nécessite 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