HELHa
Haute École Louvain en Hainaut

Activité d'intégration
UE 17
Projet

Bachelier en Informatique de gestion Mons

Trivial Pursuit : The World of Computing

Groupe n°A06

LADERRIERE Loïc & EL KOUKOUCHI Bilal & SAHIOUNI Sacha

2017 – 2018

Table des matières

I. Introduction : 3
II. Présentation du Projet : 3
III. Description générale de l'application 4
IV. Analyse 5
A. Les users stories 5
B. Diagrammes 8
1. Diagramme de classe général 8
2. Diagramme de classe détaillé 10
3. Diagramme général de la vue 14
4. Diagramme exception 17
5. Diagramme du design pattern 17
V. Code Source 18
A. Le code du design pattern 18
B. Le code du tableau utilisé en introspection 24
VI. Implémentation 32
VII. Tests Unitaires 52
VIII. Conduite de Projet 54
IX. Conclusion 57
A. Les difficultés rencontrées 57
B. Ce que le projet nous a apporté 57

I. Introduction :

Dans le cadre de notre formation, pour le cours de Projet de deuxième année, il nous a été demandé de développer une application en Java ressemblant au jeu « Trivial Pursuit ».
Nous devions développer cette application en groupe et utiliser les différentes notions techniques abordées aux différents cours, lors de notre année, comme JavaFX pour l'interface graphique de cette application.
Le but de ce projet est de nous évaluer sur nos compétences à pouvoir développer une application fonctionnelle, en respectant les différents critères qui nous ont été donnés, et notre capacité à travailler en équipe de manière « Agile ». Pour cela nous devions respecter la méthodologie de travail « Scrum », en divisant notre projet en plusieurs étapes, appelées « sprints », pour suivre au mieux l'évolution de notre projet, et pour nous assurer de ne pas nous égarer de notre objectif final. A la fin de chaque « sprint », nous devions rendre un rapport à notre professeur de toutes les actions qui ont été faites lors de celui-ci, ainsi que les actions en cours et celles à venir. Nous devions également faire part, des points positifs et négatifs rencontrés lors de chaque « sprint », et de définir les points à améliorer ainsi que des moyens pour y remédier. Après réception et lecture de notre rapport, notre professeur et notre groupe tenions un échange sur les différents points de l'avancement de notre projet, les éléments à garder et à développer ainsi que les éléments à rectifier ou supprimer. Cette méthode vise à observer notre adaptation à prendre en compte les conseils et indications donnés par notre professeur et à agir en conséquent sur notre travail.
Bien évidemment, cet exercice a aussi pour but d'approfondir nos connaissances en nous poussant à effectuer des recherches supplémentaires de manière autonome pour compléter notre application.
Ce projet comporte aussi en fin, une présentation orale en anglais, devant plusieurs professeurs pour nous mettre en situation de présentation de nos futurs projets face à d'éventuels futurs clients. Ce type d'exercice est en grande partie, un moyen de nous préparer à la réalisation de notre TFE en dernière année mais aussi à nous préparer à notre future profession.

II. Présentation du Projet :

Comme il a été dit précédemment le sujet de ce projet consiste à réaliser une application Java similaire au jeu « Trivial Pursuit ». « Trivial Pursuit » est un jeu de société axé sur la culture générale, dans ce cas, sur l'informatique. La principale difficulté de ce jeu repose sur la capacité des joueurs à répondre à différents types de questions sur six diverses catégories. En règle générale, ce jeu se joue à l'aide d'un dé ordinaire et d'un plateau en forme de cercle comportant six branches à l'intérieur. Des cases de couleurs correspondantes aux différentes catégories de questions sont réparties ainsi qu'une case placée à l'extrémité de chaque rayon pour chaque couleur. Chaque joueur met son pion correspondant à sa couleur sur sa case dédiée au début de son rayon pour commencer la partie.
Successivement, les joueurs lancent le dé et déplacent leurs pions dans la direction de leur choix jusqu'à la case indiquée par le nombre du dé. Lorsque le pion d'un joueur atteint une case, un autre joueur tire une carte correspondant à la couleur de cette case et lui pose la question écrite sur cette carte. Si le joueur donne la bonne réponse, il peut encore jouer et s'il tombe sur l'une des cases placées à l'extrémité d'un rayon, il peut remporter une part du camembert de la roue, de la couleur indiquée s'il ne l'a pas déjà eue au cours la partie. L'objectif de chaque joueur est de collectionner en premier les six parts de chaque catégorie et de placer son pion au centre du cercle pour remporter la partie.

La majeure partie des groupes devaient réaliser une application ressemblant le plus possible à ce jeu et devaient également la réaliser entièrement en anglais. Mais pour notre groupe, les règles ont été différentes. Etant donné que nous n'étions que deux, au début de ce projet. Nous devions toujours réaliser l'application en anglais mais nous n'étions pas obligés de faire un mode de jeu ayant pour support un plateau. C'est pour cela que notre jeu utilise une roue que l'on fait tourner, et qui s'arrête de manière aléatoire sur une catégorie de questions. Comme ce jeu peut être joué seul ou à plusieurs, nous avons développé des modes de jeux en conséquence. Seul, le joueur doit faire le meilleur score possible sur un nombre de manches définies et remporter toutes les parties des catégories en répondant aux questions. Le mode deux joueurs, quant à lui, est un duel opposant les joueurs à « buzzer » le premier pour répondre à la question, le premier des deux joueurs qui obtient toutes les parties des catégories remporte la partie.

III. Description générale de l'application

Comme nous l'avons dit précédemment notre jeu est un « Trivial Pursuit » particulier, il possède un mode de jeu pour un joueur, un mode pour deux joueurs qui se jouent sur un seul ordinateur. Mais notre jeu possède également un mode de jeu en réseau local, pour deux joueurs, sur deux ordinateurs différents. Le mode réseau utilise la méthode du « multithreading ».

Le développement de notre application s'est fait à travers l'utilisation de plusieurs éléments différents. Parlons dans un premier temps de l'interface graphique de notre application. L'interface graphique de notre application a été faite avec l'utilisation de « JavaFX », notamment pour la mise en place des différents panneaux, la création et la disposition de certains éléments, mais également pour les animations. Pour rendre l'interface graphique plus chaleureuse, et pour qu'elle se rapproche d'un thème lié à l'informatique, nous avons utilisé les notions de « CSS » en suppléments, pour ajouter des fonds d'écrans et pour le design de l'interface (boutons, styles d'écritures, couleurs,etc …).
Pour cette application nous avons utilisé un patron de conception, le « memento » pour permettre l'annulation d'une ou plusieurs modifications. Ce patron de conception est utilisé uniquement pour gérer le « Deck ».
Dans notre application, nous avons également utilisé la notion d'introspection, pour permettre de générer un tableau qui va adapter sa disposition en fonction de la liste qu'on lui fournit. Cela nous permet d'éviter de réécrire plusieurs fois le même code pour chaque tableau. Nous avons réalisé des tests unitaires pour vérifier le bon fonctionnement dans les classes nécessaires telles que les classes Deck, Game et Question.
Notre application possède une exception qui se manifeste lors d'un chargement d'une sauvegarde inexistante.
En règle générale, il y a une règle pour l'attribution des noms des fichiers, la première partie du nom du ficher est l'objet du fichier, et la deuxième partie est séparée par un « underscore » et donne une précision sur le fichier, comme par exemple « deck_backup », représentant un backup du « Deck ».Nous avons utilisé des librairies « GSON » pour enregistrer certaines données en fichiers « json » et « javax.mail » pour envoyer des mails lors d'une inscription d'un compte pour se connecter au jeu.

IV. Analyse

A. Les users stories

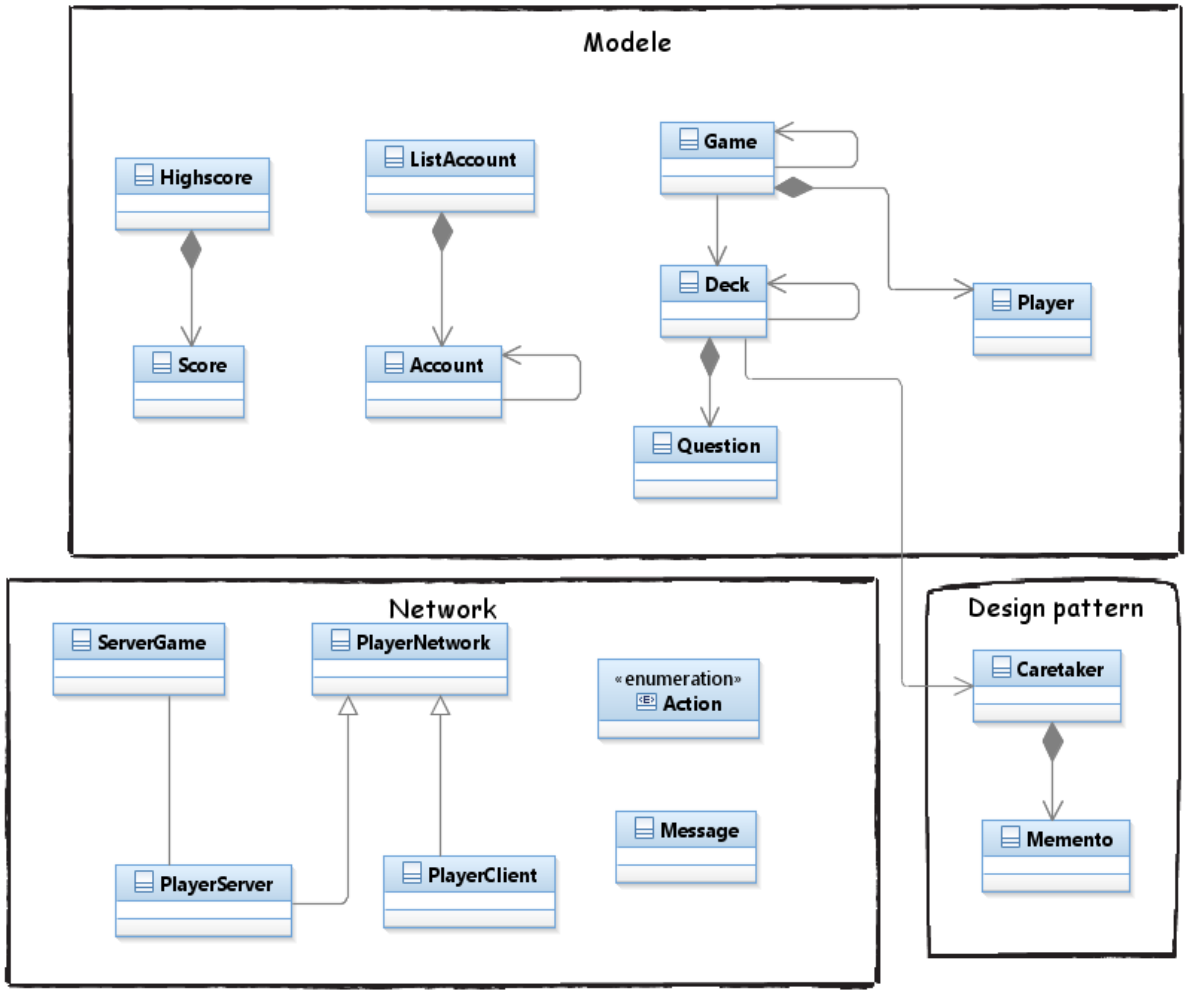
US	To Do	Doing	Done
US-01 En tant qu'utilisateur je peux lancer une partie en mode express (solo)			Lancer la roue choisit une catégorie et une question aléatoire
			La question s'affiche et un champ est disponible pour répondre
			Un timer valide le champ lorsqu'il atteint 0
			Mise en place d'un système de calculs de points pour le score
			Mise en place d'une case bonus ou d'un malus sur la roue
			Ajout d'une portion par bonne réponse pour la catégorie en question
US-02 En tant qu'administrateur je peux gérer un deck			Ajouter une question
			Supprimer une question
			Modifier une question
			Gérer les doublons lors de l'ajout
US-03 En tant qu'utilisateur je peux ouvrir un menu d'option en jeu			Le bouton Save permet de sauvegarder la partie
			Le bouton Exit permet de quitter la partie

			Le bouton Return permet de revenir au menu principal
			Le bouton Cancel permet de fermer la fenêtre
US-04 En tant qu'utilisateur je peux jouer une partie en mode duel			Lancer la roue choisit une catégorie et une question aléatoire
			Affichage d'une question pour les deux joueurs
			Mise en place d'un système de buzzer
			Mise en place d'un timer général (pour la question)
			Mise en place d'un timer pour chaque joueur
			Ajout d'une portion pour le joueur ayant bien répondu à la question de la catégorie
US-05 En tant qu'utilisateur je peux sauvegarder et reprendre une partie			Sauvegarder une partie en mode Solo
			Reprendre une partie en mode Solo
			Sauvegarder une partie en mode Duel
			Reprendre une partie en mode Duel
US-06 En tant qu'utilisateur je peux avoir un compte personnel			Connexion en tant qu'invité
			Création d'un compte
			Envoie d'un mail après inscription
			Accès d'un panneau administrateur si connecté en tant qu'administrateur
			Sauvegarde liée au compte

US-07 En tant qu'administrateur je peux gérer les comptes			Ajouter un compte
			Modifier un compte
			Supprimer un compte
US-08 En tant qu'utilisateur je peux jouer en multijoueur en réseau local			Héberger une partie
			Rejoindre une partie via l'adresse IP
			Même principe de jeu que pour le mode Duel
US-09 En tant qu'utilisateur je peux accéder à un menu pour configurer ma partie			Choix du mode de Jeu
			Paramétrer certaines variables de la partie
			Lancer la partie
US-10 En tant qu'utilisateur je veux avoir une belle interface			Interface plus colorée et uniforme
			Question entourée de la couleur de la catégorie
US-11 En tant qu'utilisateur je peux voir le résultat de ma réponse			Une alerte s'ouvre en fonction du résultat
			Un son est joué en fonction du résultat

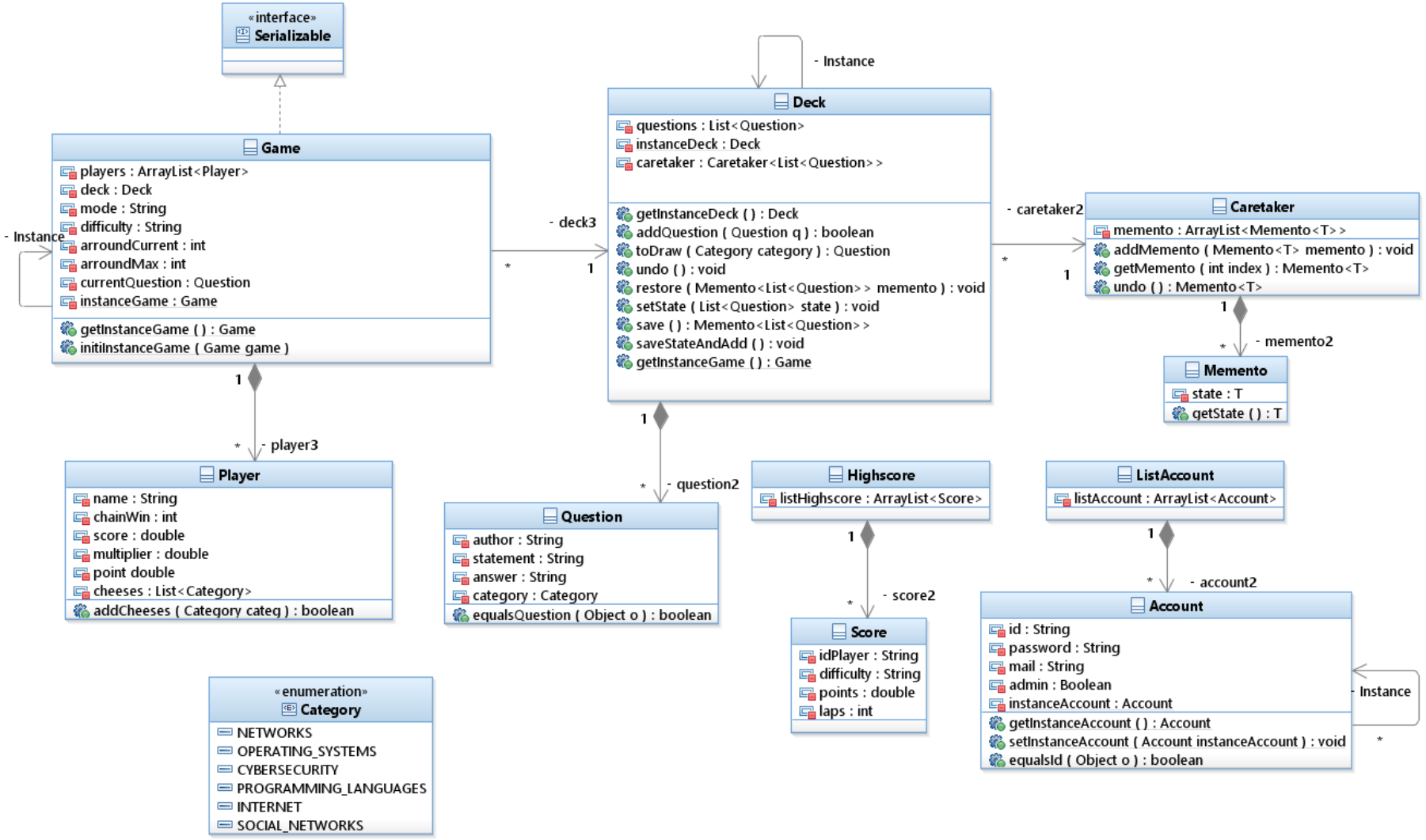
B. Diagrammes

1. Diagramme de classe général

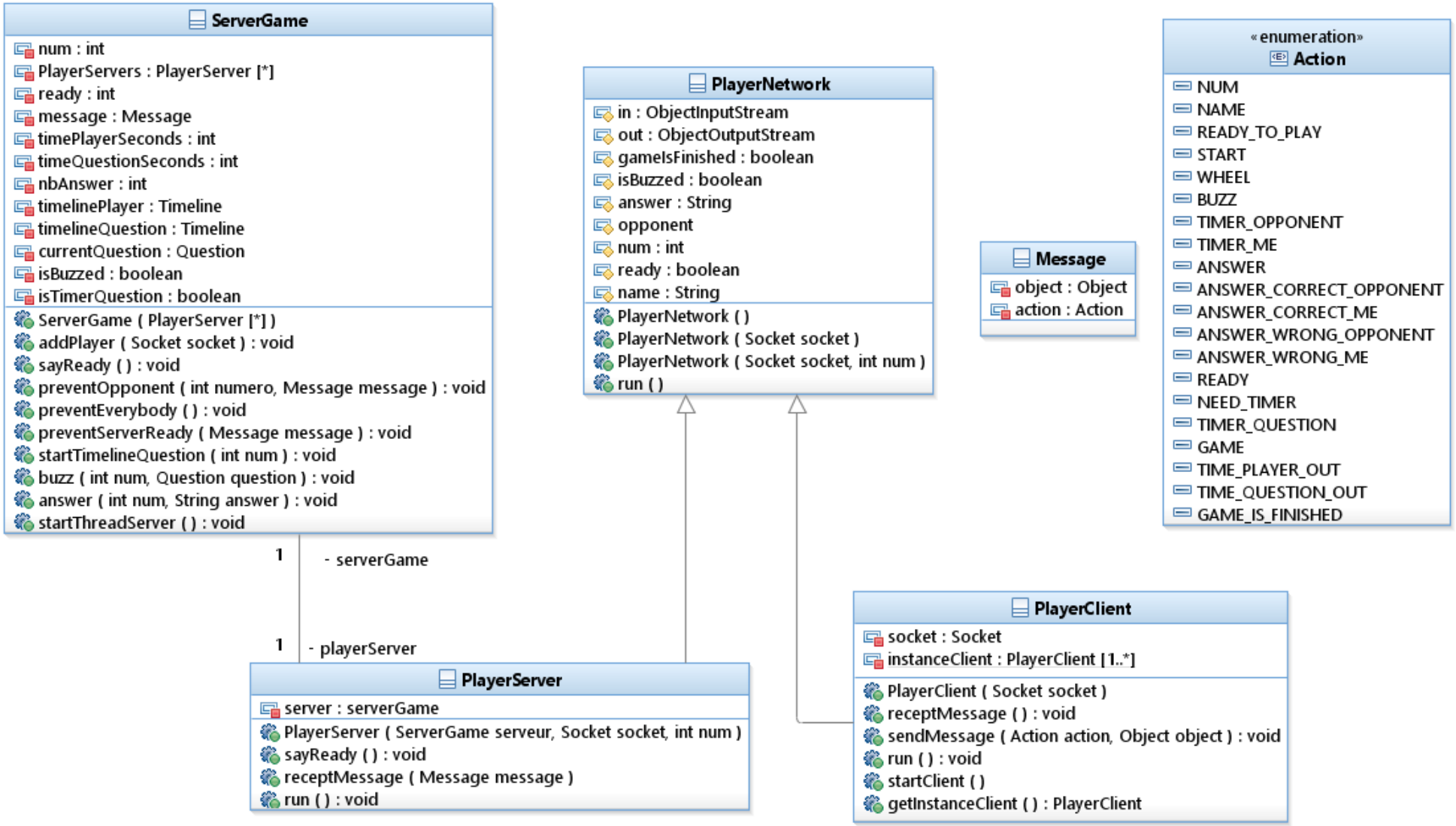


Le package « Modele » contient toute nos classes principales.
Le package « Network » contient toutes les classes pour créer, rejoindre et gérer un serveur de jeu.
Le package Design Pattern contient les classes de notre design pattern.

2. Diagramme de classe détaillé

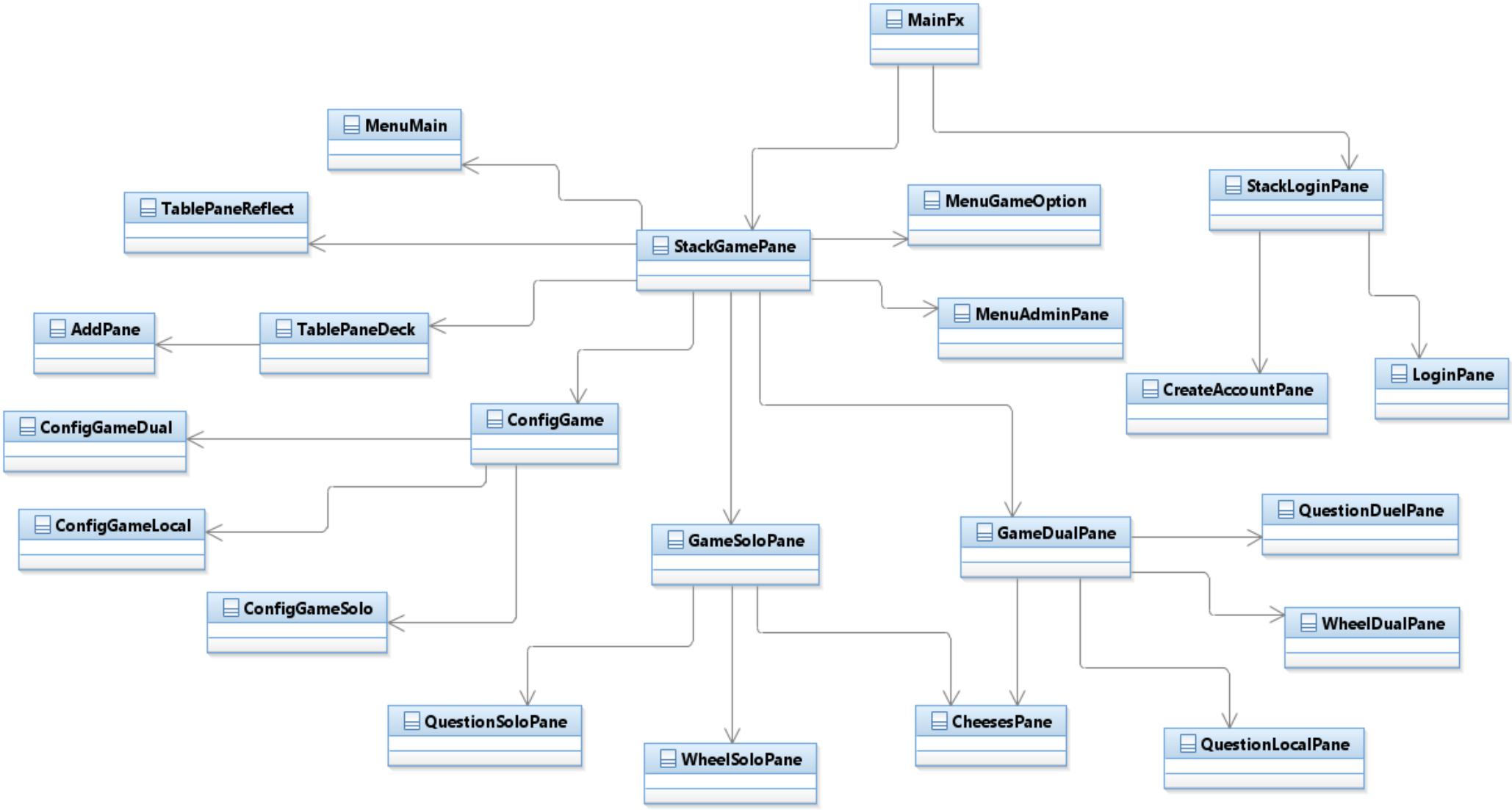


Explications des classes :
« Player » représente un joueur dans une partie.
« Game » est la classe principale. Elle contient le nécessaire afin de lancer une partie. C'est un singleton car nous ne voulions pas que le joueur puisse lancer plusieurs parties simultanément. Il implémente l'interface « Serializable » afin de gérer sa persistance.
« Deck » est une composition de question. C'est aussi un singleton. Il implémente « Serialisable » pour qu'il soit possible de le transmettre via un réseau lors d'une partie en multijoueur local.
« Highscore » est une composition de « Score ». Il est utilisé afin d'enregistrer une liste de score en un seul objet.
« Score » contient les informations d'une partie qui a été fini par un joueur. Il a pour but d'être affiché dans un tableau de score.
« ListAccount » est une composition de « Account ». Il est utilisé afin d'enregistrer une liste de account en un seul objet.
« Account » est un singleton car nous voulions empêcher la connexion de plusieurs comptes simultanément.


ServerGame
num : int
PlayerServers : PlayerServer [*]
ready : int
message : Message
timePlayerSeconds : int
timeQuestionSeconds : int
nbAnswer : int
timelinePlayer : Timeline
timelineQuestion : Timeline
currentQuestion : Question
isBuzzed : boolean
isTimerQuestion : boolean
ServerGame (PlayerServer [*])
addPlayer (Socket socket) : void
sayReady ( ) : void
preventOpponent (int numero, Message message) : void
preventEverybody ( ) : void
preventServerReady (Message message) : void
startTimelineQuestion (int num) : void
buzz (int num, Question question) : void
answer (int num, String answer) : void
startThreadServer ( ) : void
1
- serverGame
1
- playerServer
PlayerNetwork
in : ObjectInputStream
out : ObjectOutputStream
gameIsFinished : boolean
isBuzzed : boolean
answer : String
opponent
num : int
ready : boolean
name : String
PlayerNetwork ( )
PlayerNetwork (Socket socket)
PlayerNetwork (Socket socket, int num)
run ( )
Message
object : Object
action : Action
«enumeration»
Action
NUM
NAME
READY_TO_PLAY
START
WHEEL
BUZZ
TIMER_OPPONENT
TIMER_ME
ANSWER
ANSWER_CORRECT_OPPONENT
ANSWER_CORRECT_ME
ANSWER_WRONG_OPPONENT
ANSWER_WRONG_ME
READY
NEED_TIMER
TIMER_QUESTION
GAME
TIME_PLAYER_OUT
TIME_QUESTION_OUT
GAME_IS_FINISHED
PlayerServer
server : serverGame
PlayerServer (ServerGame serveur, Socket socket, int num)
sayReady ( ) : void
receptMessage (Message message)
run ( ) : void
PlayerClient
socket : Socket
instanceClient : PlayerClient [1..*]
PlayerClient (Socket socket)
receptMessage ( ) : void
sendMessage (Action action, Object object) : void
run ( ) : void
startClient ( )
getInstanceClient ( ) : PlayerClient

« ServerGame » est le serveur de jeu. Il va gérer la connexion des joueurs, la vérification des réponses, les différents timer, etc…
« PlayerNetwork » a ses variables en « protected » afin qu'elle puissent être uniquement utilisé par toutes les autres classes du package « network ».
« PlayerServer » permet à « ServerGame » de recevoir des messages en écoutant en boucle le port du pc via un thread. Il hérite de la classe « PlayerNetwork ».
« PlayerClient » permet à un joueur d'envoyer un message au serveur ou d'en recevoir via un thread. Il hérite de la classe « PlayerNetwork ».
« Message » permet d'uniformiser une discussion. La variable « object » représente le contenu du message et la variable « action » permet au destinataire de savoir ce qu'il doit faire avec ce contenu.
« Action » est une énumération définissant toutes les actions que peuvent réaliser un serveur ou un client.

3. Diagramme général de la vue

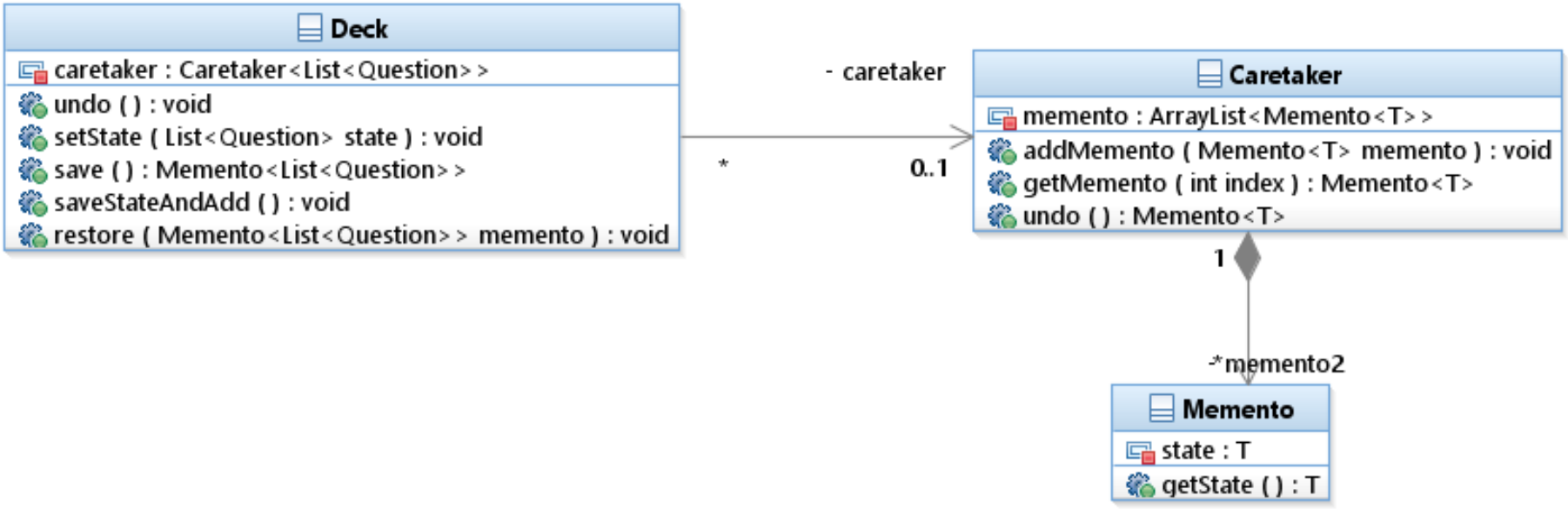


Explication des classes :
« StackGameLoginPane » » est un StackPane utilisé pour stocker « CreateloginPane » et « LoginPane » afin de faciliter leur accès.
« CreateLoginPane » est un GridPane afin de faciliter la disposition des boutons en forme de grille.
Il contient des champs à remplir ainsi que des labels qui leurs sont associés pour entrer les informations nécessaires pour créer un compte. Ainsi qu'un bouton pour revenir au menu et un autre pour valider la création de son compte.
« StackGamePane » est un StackPane utilisé pour stocker les autres classes afin de rendre leur accès plus facile.
« MenuMain » est un GridPane afin de faciliter la disposition des boutons en forme de grille. C'est le menu principal de notre jeu.
« TablePaneReflect » est BorderPane afin de faciliter la disposition d'élément dans les divers coins de la fenêtre. Il est composé d'un tableau utilisant l'introspection. Cette classe est visible dans le menu « HighScore » et « Account ».
« TablePaneDeck » est BorderPane afin de faciliter la disposition d'élément dans les divers coins de la fenêtre. Il est composé d'un tableau servant à afficher la liste des questions contenu dans un deck. Dans la partie supérieure se trouve un autre pane (AddPane). Dans la partie inférieure se trouve 4 boutons permettant la sauvegarde du deck, la suppression d'une question, un retour au menu principal et l'annulation d'une modification dans le tableau.
« AddPane » est un GridPane afin de faciliter la disposition des boutons en forme de grille. Il permet l'ajout d'une question dans le tableau de « TablePaneDeck ».
« ConfigGame » est BorderPane afin de faciliter la disposition d'élément dans les divers coins de la fenêtre. C'est un menu permettant de configurer une partie. En haut, se trouve un bouton permettant de changer le mode de jeu. Il est composé de 3 autres panes. Chacun représentant un sous-menu permettant de configurer une partie en fonction du mode de jeu.
« GameSoloPane » est BorderPane afin de faciliter la disposition d'élément dans les divers coins de la fenêtre. C'est un pane regroupant d'autres panes afin de former une interface pour le mode solo. Il est composé de « QuestionSoloPane », « WheelSoloPane » et de « CheesesPane ».
« QuestionSoloPane » est un StackPane. Il affiche la question ainsi qu'un champ pour y répondre.
« WheelSoloPane » est un StackPane contenant une image de la roue. Un bouton se trouve au centre afin de la faire tourner et d'afficher « QuestionSoloPane ».
« CheesesPane » est un GridPane afin de faciliter la disposition des boutons en forme de grille. Il contient une image de toutes les portions de catégories. Chacune s'illumine lorsque le joueur la gagne.
« GameDualPane » est BorderPane afin de faciliter la disposition d'élément dans les divers coins de la fenêtre. C'est un pane regroupant d'autres panes afin de former une interface pour le mode duel. Il est composé de « QuestionDuelPane » si le mode de jeu est « Duel » ou bien « QuestionLocalPane » si le mode de jeu est « Local ». Ainsi que de « WheelDualPane » et de « CheesesPane ».

« QuestionDualPane » est un StackPane. Il affiche la question ainsi qu'un champ par joueur pour y répondre. Il réagit au touche « MAJ » et « UP » pour que chaque joueurs puissent buzzer et répondre.
« QuestionLocalPane » est un StackPane. Il affiche la question ainsi qu'un champ par joueur pour y répondre. Il réagit au touche « MAJ » pour que chaque joueurs puissent buzzer et répondre.
« WheelDualPane » est un StackPane contenant une image de la roue. Un bouton se trouve au centre afin de la faire tourner et d'afficher « QuestionDualPane ».

4. Diagramme exception

5. Diagramme du design pattern



V. Code Source

A. Le code du design pattern

Nous avons choisi le Memento comme design pattern car il est idéal dans la gestion d'éléments car il permet d'annuler plusieurs actions telles que la suppression ou la modification d'éléments. Il s'intègre donc bien dans le tableau gérant notre deck.
On peut prendre comme exemple que l'utilisateur a fait plusieurs modifications sur le deck et qu'il désire annuler seulement une modification, il ne sera pas obligé d'annuler ses autres modifications.

```
public class Memento<T> {


	private T state;

	/**
	 * @param state
	 * The state to save
	 */
    public Memento(T state) {
        this.state = state;
    }
    /**
        * @return the state saved
        */
    public T getState() {
        return state;
    }

	@Override
	public String toString() {
		return "\nMemento [state=" + state + "]";
	}
}
```

```java
public class Caretaker<T> {

	private ArrayList<Memento<T>> mementos;

	public Caretaker(){
		mementos = new ArrayList<>();
	}

	/**
	 * Hold a memento in memory
	 * @param memento to save
	 */
	public void addMemento(Memento<T> memento) {
		mementos.add(memento);
	}

	/**
	 * @param index of the position of the memento to return
	 * @return the memento at the specified position in this list.
	 */
	public Memento<T> getMemento(int index) {
		return mementos.get(index);
	}

	/**
	 * Returns the previous memento and remove the actual state of the memory
	 * @return the previous memento
	 */
	public Memento<T> undo(){
		if(mementos.size()>1){
			mementos.remove(mementos.size()-1);
			return mementos.get(mementos.size()-1);
		}
		return mementos.get(0);
	}

	@Override
```

```
	public String toString() {
		return "Caretaker [mementos=" + mementos + "]";
	}

}

public class Deck implements Serializable {

	private static final long serialVersionUID = 1L;

	private List<Question> questions;
	private static Deck instanceDeck;
	private static Caretaker<List<Question>> caretaker;

	/**
	 * It allows to create a deck
	 */
	public Deck() {
		this.questions = new ArrayList<Question>();
	}
	/**
	 * @return an instance of the deck
	 */
	public static Deck getInstanceDeck() {
		if(instanceDeck == null){
			instanceDeck = new Deck();
			caretaker = new Caretaker<>();
		}
		return instanceDeck;
	}
	/**
	 * add a question if the deck don't contain it
	 * @param q
	 * The question that needs to be added
	 * @return The resultat
	 */
	public boolean addQuestion(Question q) {
```

```
		for (Question question : questions) {
			if(q.equalsQuestion(question)) {
				return false;
			}
		}
		this.questions.add(q);
		saveStateAndAdd();
		return true;
	}
	/**
	 * @return The list of the questions
	 */
	public List<Question> getQuestions() {
		return questions;
	}
	/**
	 * @param questions
	 * The new list of the question
	 */
	public void setQuestions(List<Question> questions) {
		this.questions = questions;
	}
	/**
	 * @return a string of the deck
	 */
	@Override
	public String toString() {
		return "Deck [questions=" + questions + "]";
	}
	/**
	 * @param category
	 * The category of the question to be drawed
	 * @return the question
	 */
	public Question toDraw(Category category) {
		int index=0;
		boolean test=false;
```

```
        Question tmp;
        Question question;
        do {
                question=this.questions.get(index);

                //Move the card drawn at the end of the deck
                if(question.getCategory()==category){
                        tmp=question;
                        this.questions.remove(index);
                        this.questions.add(tmp);
                        test=true;
                }
                index++;
        } while (test==false);
        return question;
}

//Memento
public void undo(){
        restore(caretaker.undo());
}
/**Methode Memento
 * restore the state from the memento
 * @param memento who contains the state to restore
 */
public void restore(Memento<List<Question>> memento) {
        setState(memento.getState());
}

/**Methode Memento
 * set the state
 * @param state
 * The new state
 */
public void setState(List<Question> state) {
        this.questions = state;
}
```

```
	/**Methode Memento
	 * save the state
	 * @return a memento who contains the saved state
	 */
	public Memento<List<Question>> save() {
		List<Question> list = new ArrayList<>();
		list.addAll(this.questions);
		return new Memento<>(list);
	}

	/**Methode Memento
	 * add the state into a memento
	 */
	public void saveStateAndAdd() {
		caretaker.addMemento(save());
	}

}
```

B. Le code du tableau utilisé en introspection

Le code qui suit est celui d'un tableau utilisé en introspection, il permet de générer un tableau qui s'adapte à la liste qu'on lui passe, il va générer les colonnes en fonction des méthodes qui sont dans la classe des objets se trouvant dans la liste (exemple : la classe « compte » possède une méthode getId() , il va créer une colonne en récupérant le nom de cette méthode et en supprimant le mot get pour garder uniquement le mot Id, et il va rendre cette éditable , et appliquer ceci a toutes les méthodes de cette classe). Nous avons utilisé l'introspection pour éviter de réécrire du code redondant pour chaque tableau.

```
public class TablePaneReflect extends BorderPane {
	private Button btnSave,btnDelete, btnBack;
	private AddPane addPane;
	private TableView table;
	private ArrayList<?> data = new ArrayList<>();
	private ObservableList<?> observableList ;
	private int i=0;
	private Object objectParent;
	private boolean isEditable;

	/**
	 * It allows to create a view with the table
	 * @param arrayList
	 * is the list of object in the table
	 * @param object
	 * is the object in which the varibale arraylist is declared
	 * @param editable
	 * defined if the table is editable
	 */
	public TablePaneReflect(ArrayList arrayList, Object object, Boolean editable) {
		data = arrayList;
		objectParent = object;
		isEditable = editable;
		HBox hbx = new HBox();

		//If the talbe is editable. Add the button "Save" and "Delete"
```

```
        if(Account.getInstanceAccount().isAdmin() == true) {
                hbx.getChildren().addAll(getBtnSave(), getBtnDelete());
        }

        hbx.getChildren().add(getBtnBack());
        hbx.setAlignment(Pos.CENTER);

        this.setCenter(getTableView());
        this.setBottom(hbx);
}

/**
 * Method TableView
 * @return a table containing an list of objects
 */
public TableView<?> getTableView(){
        if(table == null) {
                //Creates an observable list from the original list
                observableList = FXCollections.observableArrayList(data);

                table = new TableView(observableList);

                //Set the table on editable if parameter is true
                table.setEditable(isEditable);

                //Set the table to select multiple objects at the same time
                table.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);

                //Get the class of an object in the list
                Class c = data.get(0).getClass();

                //For each methods of the object of the list
                for (Method method : c.getMethods()) {

                        //Get the name of the method
                        String nameMethod = method.getName();
```

```
//If this name start by "getClass" or "getInstance" -> ignore this method
if (!nameMethod.startsWith("getClass") && !nameMethod.startsWith("getInstance")) {
	String columnName = null;
	if (nameMethod.startsWith("is")) {
		columnName = nameMethod.replace("is", "");
		String nameSetMethod = nameMethod.replace("is", "set");

		//Create the column and add it in the table
		addColumn(columnName, data.get(0));

		//Get this column
		TableColumn <Object,Boolean> column = (TableColumn) table.getColumns().get(i);

		//Convert the column to boolean (Because to edit. The method created have to return a boolean)
		column.setCellFactory(TextFieldTableCell.forTableColumn(new BooleanStringConverter()));

		//Allow to edit a table cell
		column.setOnEditCommit((CellEditEvent<Object,Boolean> event) -> {

			TablePosition<Object,Boolean> pos = event.getTablePosition();

			Boolean newValue = event.getNewValue();

			int row = pos.getRow();
			Object objTable = event.getTableView().getItems().get(row);

			objTable = data.get(0).getClass().cast(objTable);

			Object[] param = {newValue};

			//Create a method that will edit the object to be modified
			Utility.callMethod(objTable, nameSetMethod, param, boolean.class);
		});
		i++;

		//Same thing but if the method is a get
	}if (nameMethod.startsWith("get")) {
```

```
columnName = nameMethod.replace("get", "");
String nameSetMethod = nameMethod.replace("get", "set");
addColumn(columnName, data.get(0));

//If the method return a "String"
if(method.getGenericReturnType() == String.class) {
	TableColumn <Object,String> column = (TableColumn) table.getColumns().get(i);

	//No need to convert because the colum is already a "String"
	column.setCellFactory(TextFieldTableCell.forTableColumn());
	column.setOnEditCommit((CellEditEvent<Object,String> event) -> {

		TablePosition<Object,String> pos = event.getTablePosition();

		String newValue = event.getNewValue();

		int row = pos.getRow();
		Object objTable = event.getTableView().getItems().get(row);

		objTable = data.get(0).getClass().cast(objTable);

		Object[] param = {newValue};

		//Create a method that will edit the object to be modified
		Utility.callMethod(objTable, nameSetMethod, param, String.class);
	});

	//If the method return a "Double"
}else if(method.getGenericReturnType() == double.class) {
	TableColumn <Object,Double> column = (TableColumn) table.getColumns().get(i);

	//Convert the column to "double" (Because to edit. The method created have to return a
"double")

	column.setCellFactory(TextFieldTableCell.forTableColumn(new DoubleStringConverter()));
	column.setOnEditCommit((CellEditEvent<Object,Double> event) -> {

		TablePosition<Object,Double> pos = event.getTablePosition();
```

```
                Double newValue = event.getNewValue();

                int row = pos.getRow();
                Object objTable = event.getTableView().getItems().get(row);

                objTable = data.get(0).getClass().cast(objTable);

                Object[] param = {newValue};

                Utility.callMethod(objTable, nameSetMethod, param, double.class);
            });

            //If the method return a "Int"
        }else if(method.getGenericReturnType() == int.class) {
            TableColumn <Object,Integer> column = (TableColumn) table.getColumns().get(i);

            //Convert the column to "int" (Because to edit. The method created have to return a
"int")
            column.setCellFactory(TextFieldTableCell.forTableColumn(new IntegerStringConverter()));
            column.setOnEditCommit((CellEditEvent<Object,Integer> event) -> {

                TablePosition<Object,Integer> pos = event.getTablePosition();

                Integer newValue = event.getNewValue();

                int row = pos.getRow();
                Object objTable = event.getTableView().getItems().get(row);

                objTable = data.get(0).getClass().cast(objTable);

                Object[] param = {newValue};

                //Create a method that will edit the object to be modified
                Utility.callMethod(objTable, nameSetMethod, param, int.class);
            });
        }
```

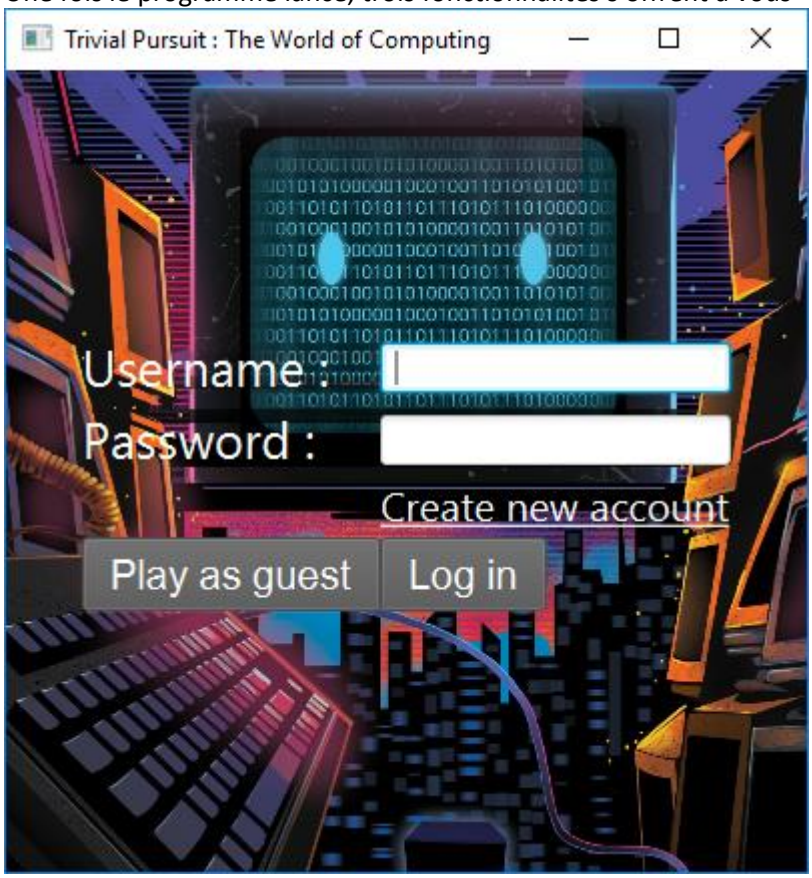
```
                                i++;
                            }

                        }
                    }
            }
            return table;
}

/**
 * Method to create a column and put it in the table
 * @param columnName
 * Name of the column
 * @param obj
 * Object containing the list
 */
public void addColumn(String columnName, Object obj) {

            TableColumn<Object,Object> column = new TableColumn(columnName);
            column.setCellValueFactory(new PropertyValueFactory<>(columnName));

            table.getColumns().add(column);
}
//Button to save the deck
public Button getBtnSave() {
            if(btnSave==null) {
                        btnSave=new Button();
                        btnSave.setText("Save");
                        btnSave.setOnAction(new EventHandler<ActionEvent>() {

                                    @Override
                                    public void handle(ActionEvent event) {
                                                // TODO Auto-generated method stub
                                                ArrayList newList = new ArrayList();
                                                for( Object o : observableList) {
                                                            newList.add(o);
                                                }
```

```
                        Object[] obj = {newList};

                        for (Method method : objectParent.getClass().getMethods()) {
                              String nameMethod = method.getName();
                              if (nameMethod.startsWith("setList")) {
                                    Utility.callMethod(objectParent, nameMethod, obj, ArrayList.class);

                              }

                        }
                        Serialisation.saveFile(objectParent.getClass().getSimpleName()+"_save", objectParent);

                  }
            });
      }
      return btnSave;
}

//Button to delete a question
public Button getBtnDelete() {
      if(btnDelete == null) {
            btnDelete = new Button("Delete");
            btnDelete.setOnAction(new EventHandler<ActionEvent>() {

                  @Override
                  public void handle(ActionEvent event) {
                        // TODO Auto-generated method stub
                        table.getItems().removeAll(table.getSelectionModel().getSelectedItems());
                  }

            });
      }
      return btnDelete;
}
//Button to back to the main menu
public Button getBtnBack() {
      if(btnBack==null) {
```

```
                btnBack=new Button();
                btnBack.setText("Back");
                btnBack.setOnAction(new EventHandler<ActionEvent>() {

                        @Override
                        public void handle(ActionEvent event) {
                                // TODO Auto-generated method stub
                                MainFx.getStackGamePane().hideAll();
                                MainFx.getStackGamePane().getMainMenu().setVisible(true);
                        }
                });
        }
        return btnBack;
    }
}
```

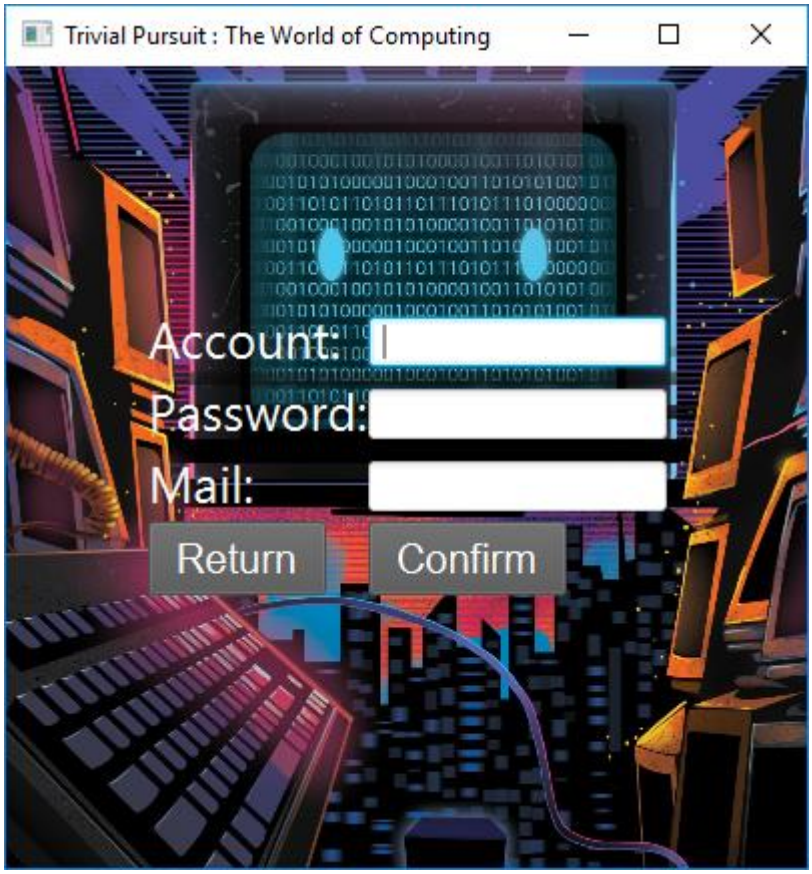
VI. Implémentation

Une fois le programme lancé, trois fonctionnalités s'offrent à vous

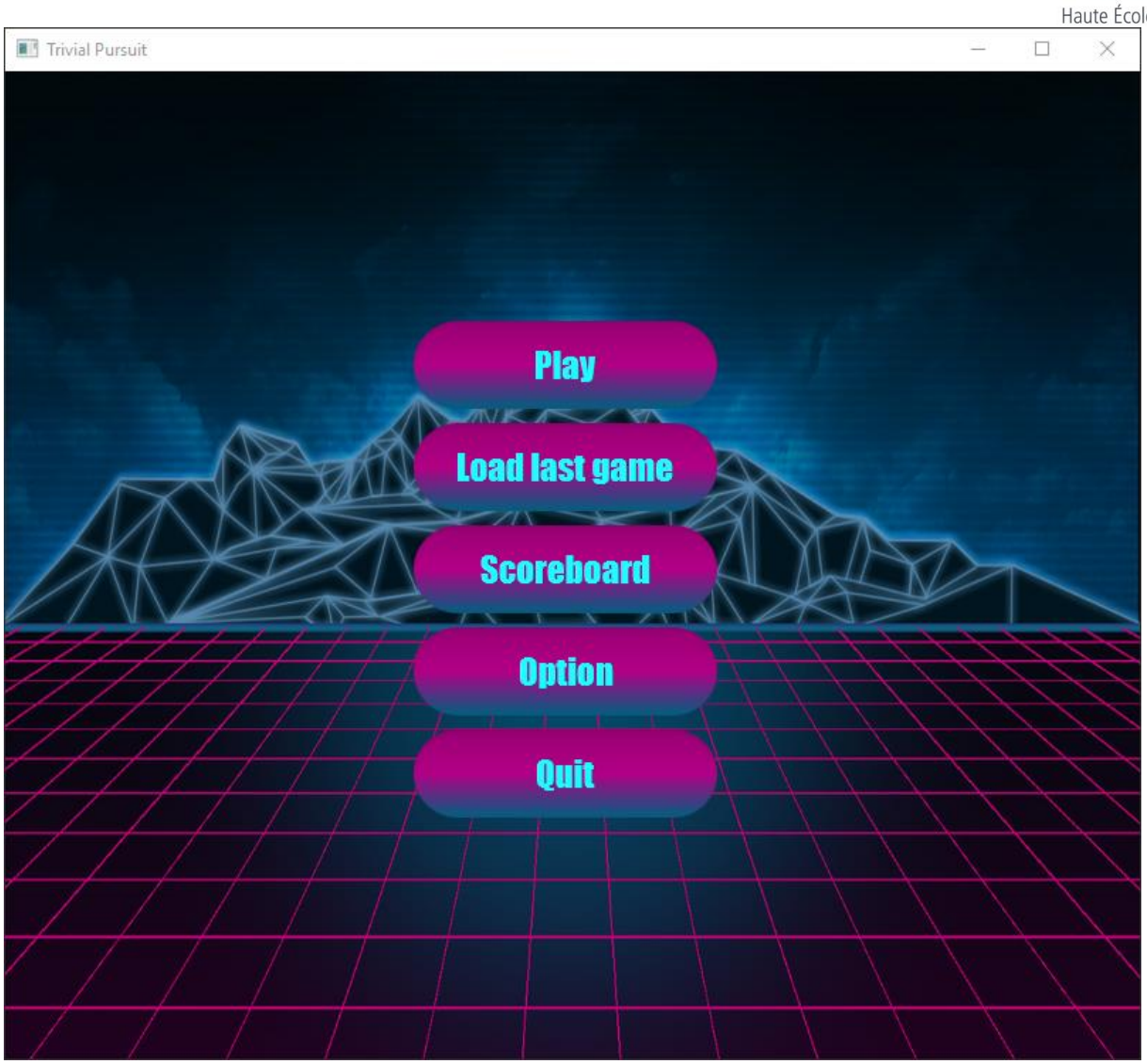


La zone de texte vous permet d'entrer votre "login" et votre "password".
Le bouton "Play as guest" vous permet de vous connecter en tant qu'invité.
Le bouton "Create new account" vous permet de vous inscrire au jeu.
Le bouton "Log in" vous permet de vous connecter en tant qu'utilisateur après avoir tapé votre "login" et votre "password" dans la zone de texte.

Après avoir cliqué sur le bouton " Create new account ", vous pourrez vous inscrire afin de devenir utilisateur sur cette page.

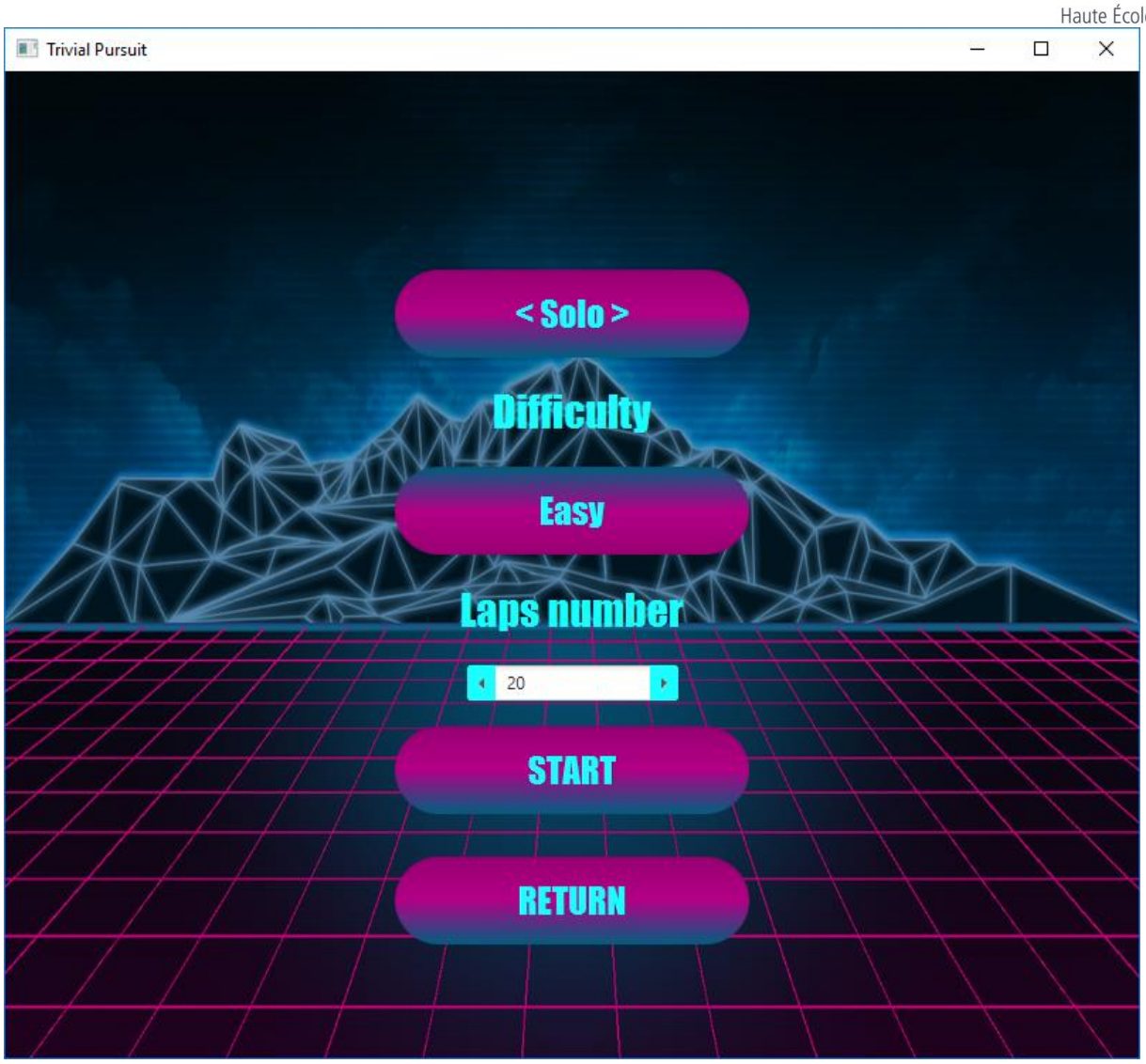


La zone de texte vous permet de créer un nouveau "login" et "password " avec votre "mail".
Le bouton "return" vous permet de revenir au menu de connexion.
Le bouton "Confirm" vous permet de sauvegarder votre enregistrement au jeu après avoir complété les différentes zones demandées.
Une fois inscription terminée, si le champ mail a été rempli, un mail sera envoyé à cette adresse.
Une fois connecté au jeu en tant qu'utilisateur ou invité, vous tomberez sur cette page.



Le bouton "Play" vous permet d'entrer dans le menu du jeu.
Le bouton "Load last game" vous permet de charger la dernière partie sauvegardée.
Si aucune partie n'a été trouvée, l'utilisateur est alerté.
Le bouton "ScoreBoard" vous permet de voir le score de tous les utilisateurs enregistrés.
Le bouton "Option" vous permet de modifier les options de jeu.
Le bouton "Quit" vous permet de fermer le jeu.

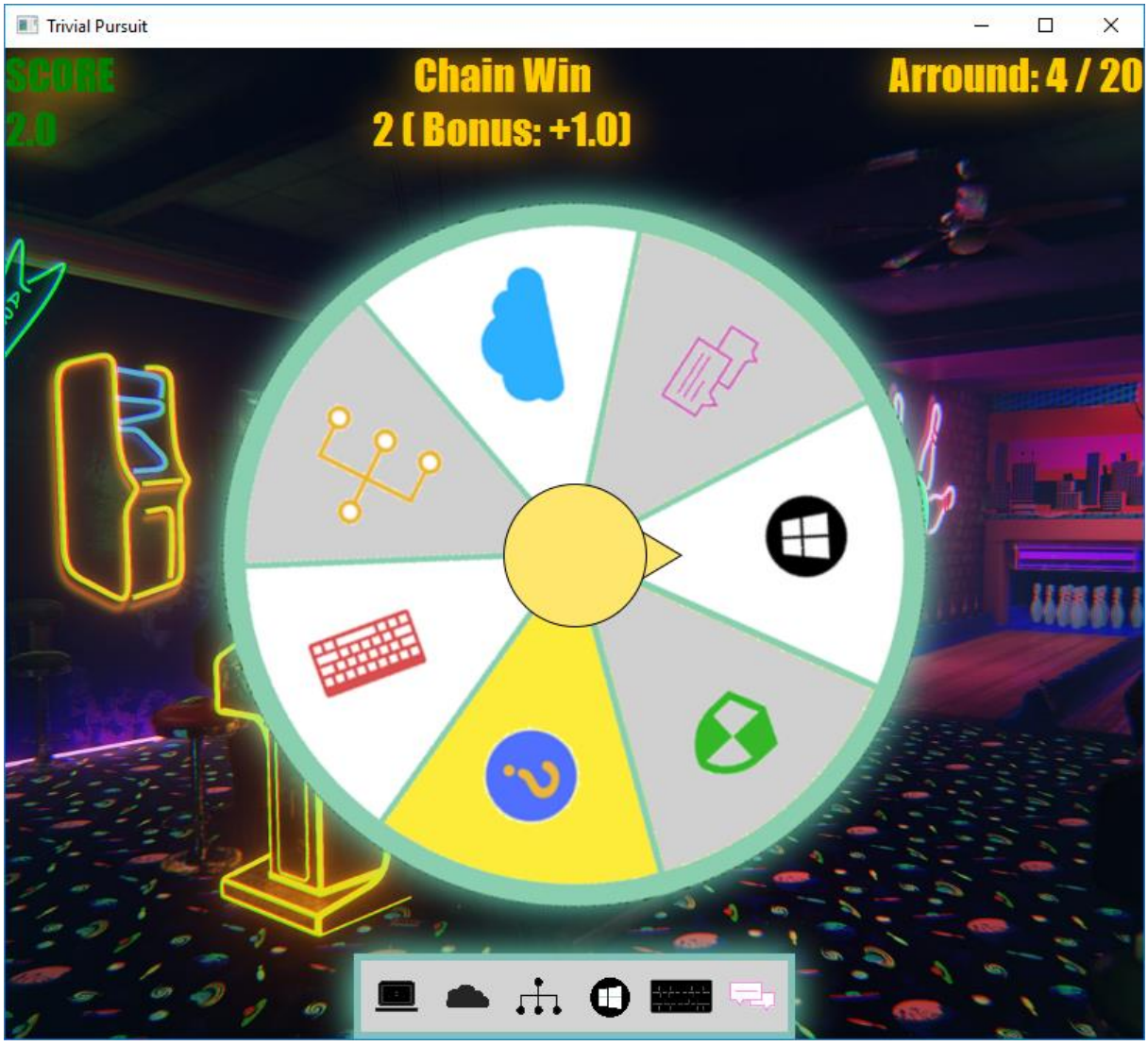
Si vous avez cliqué sur le bouton "Play", vous serez envoyé sur cette page.



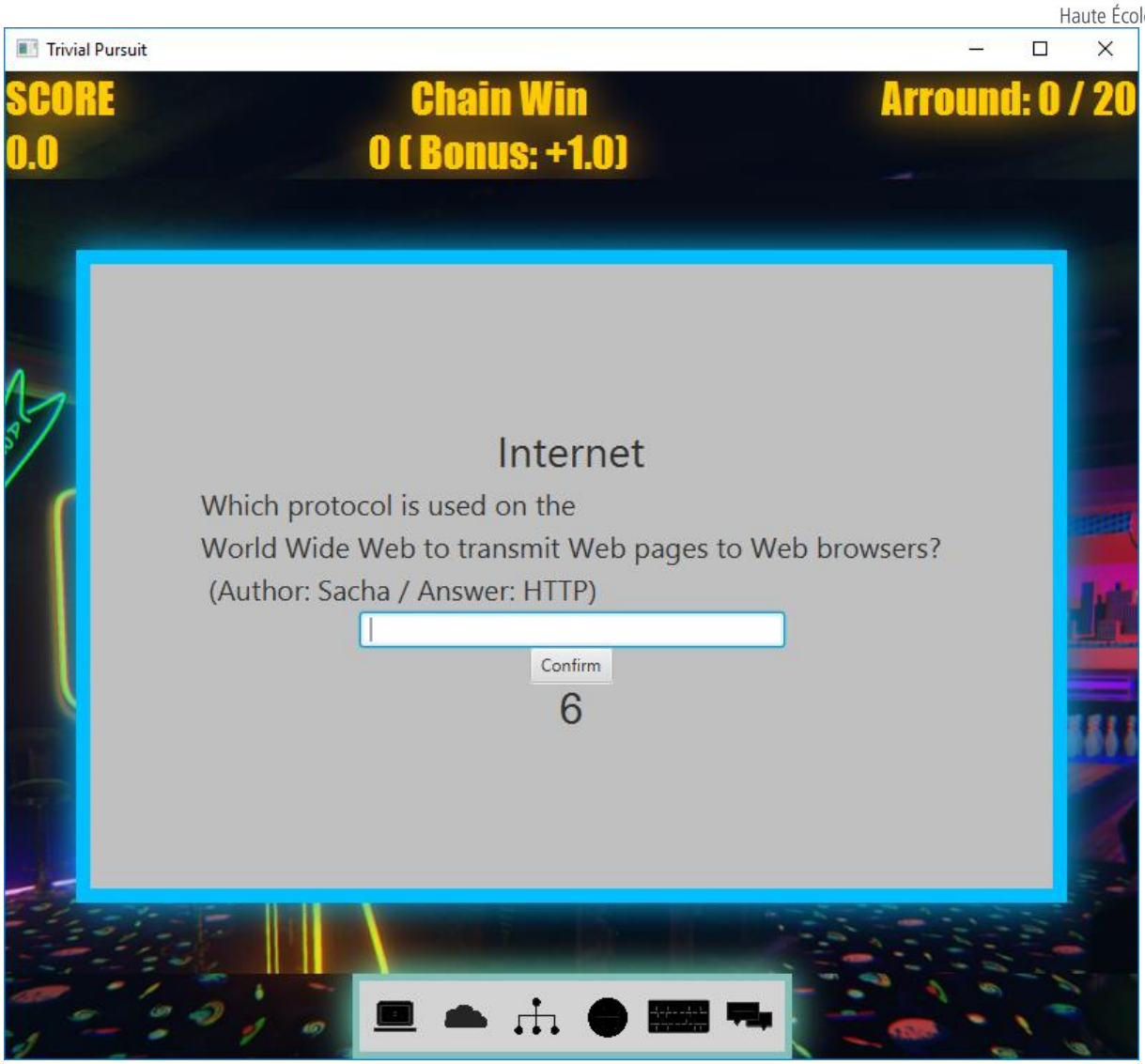
Le premier bouton vous permet de choisir votre mode de jeu (Mode solo par défaut).
Le deuxième bouton vous permet de choisir la difficulté de la partie.
La zone "Laps number" vous permet de sélectionner le nombre de manche de la partie.
Le bouton "Start" vous permet de commencer la partie.
Le bouton "Return" vous permet de revenir au menu principal.

Après avoir choisi le mode solo, vous démarrez le jeu.

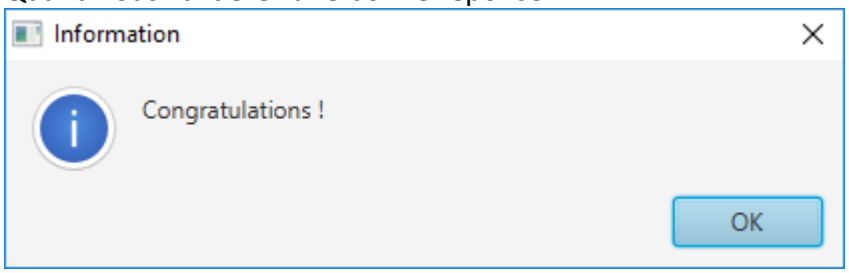
La zone "Score" vous permet de voir votre score maximal atteint durant la partie.
La zone "Chain win" vous permet de voir le nombre de bonnes réponses enchainées.
La zone "Arround" vous permet de voir la manche de la partie.
La zone en bas vous permet de voir le nombre de camemberts gagnés durant la partie. Il s'allumera une fois que vous aurez bien répondu à son thème. La partie sera donc finie quand vous aurez récupéré tous les camemberts ou après avoir atteint la dernière manche.

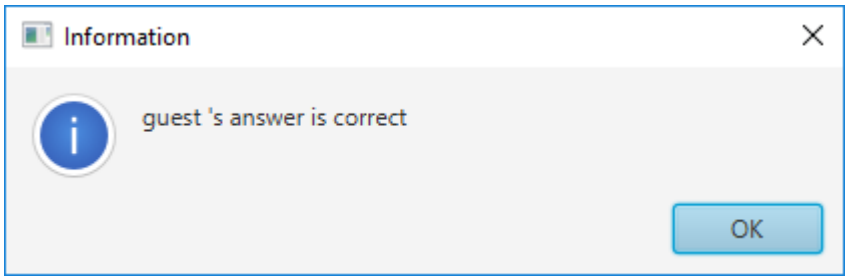


Pour démarrer la roue, il suffit de cliquer au centre, vous tomberez alors soit sur un thème ou un bonus ou malus (qui vous fera gagner ou perdre des points).
Une fois un thème choisi vous tomberez sur cette page.

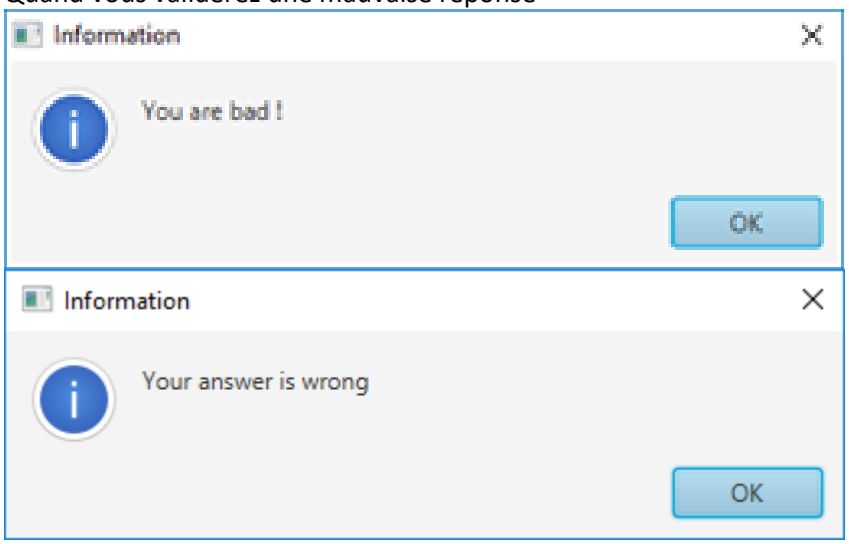


Vous verrez alors votre thème suivi de sa question.
La zone de texte vous permettra de répondre à sa question. Pour confirmer la réponse il faut appuyer sur la touche de votre clavier "enter" ou sur le bouton "Confirm".
Il y a un délai pour répondre à la question.
3 possibilités sont alors possibles (pour tous les modes) :
Vous recevrez une alerte où il faudra appuyer sur "ok" pour reprendre la partie pour chacun des cas.
Quand vous validerez une bonne réponse.

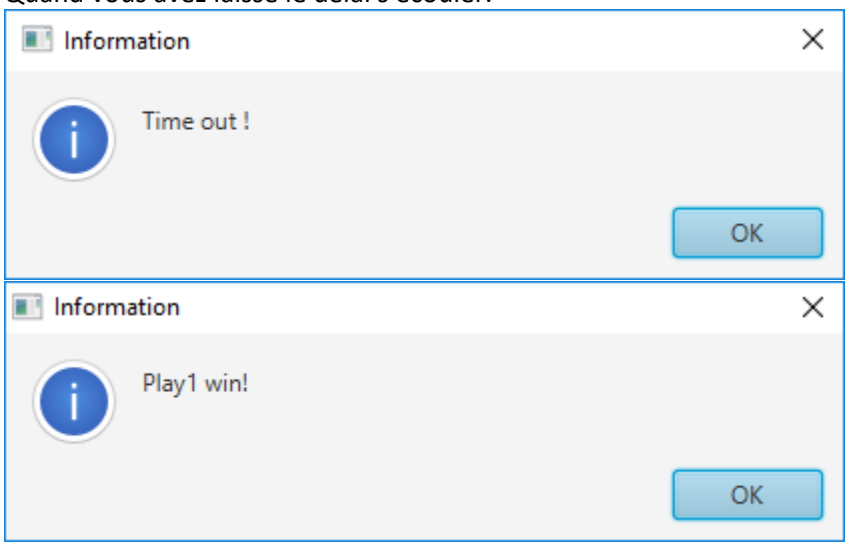




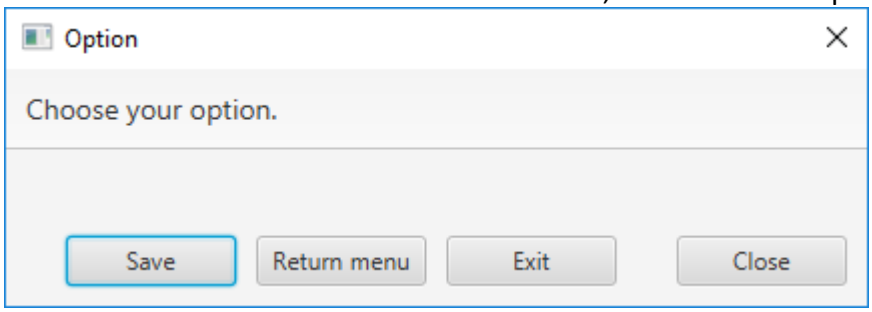
Quand vous validerez une mauvaise réponse



Quand vous avez laissé le délai s'écouler.



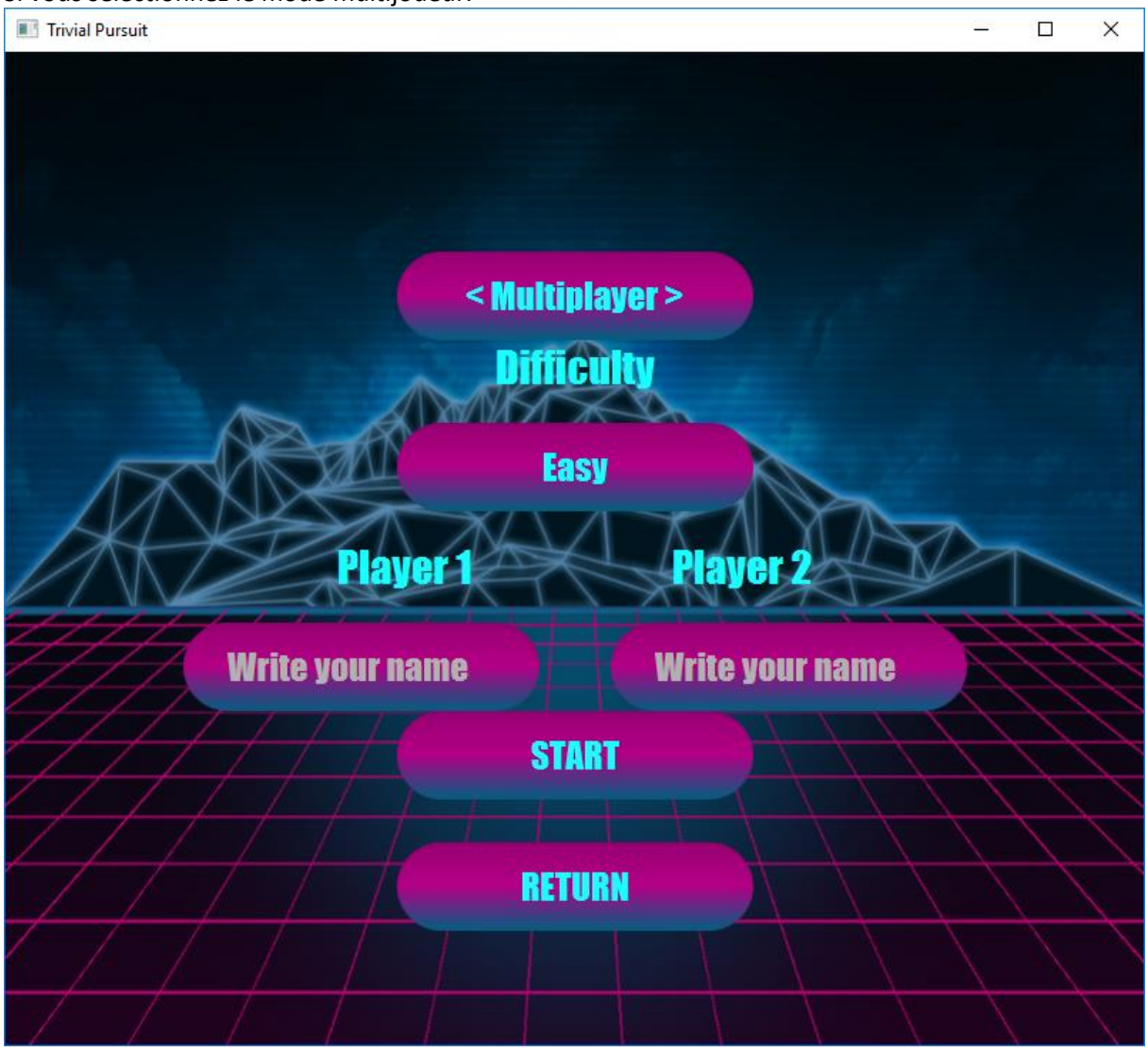
Si vous utilisez la touche de votre clavier "Esc", vous ouvrirez l'option en jeu.



Le bouton "Save" vous permet de sauvegarder votre partie pour la reprendre plus tard.

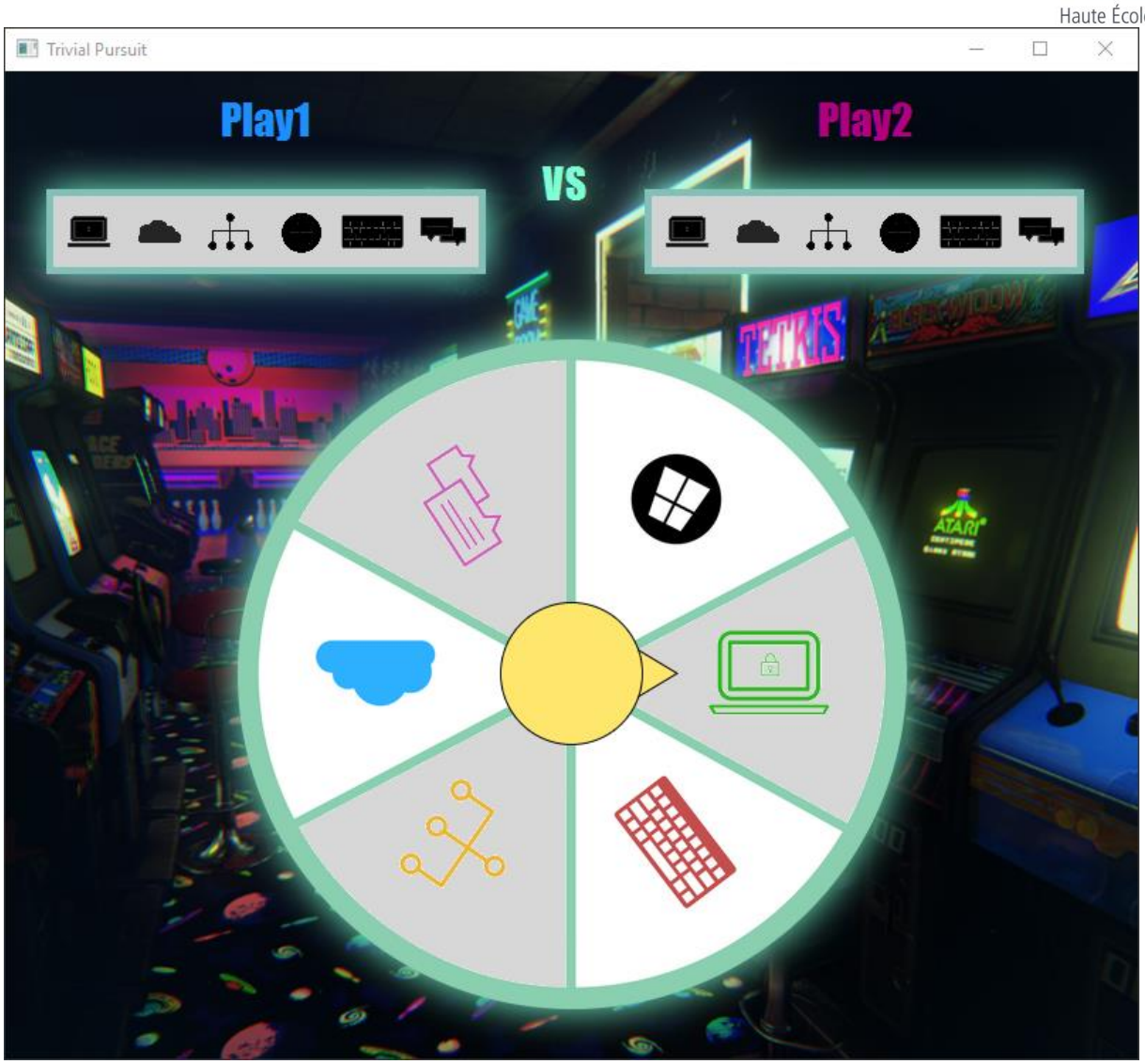
Le bouton "Return menu" vous permet de revenir au menu principal.
Le bouton "Exit" vous permet de fermer complètement le jeu.
Le bouton "Close" vous permet de fermer la page d'options.

Si vous sélectionnez le mode multijoueur.



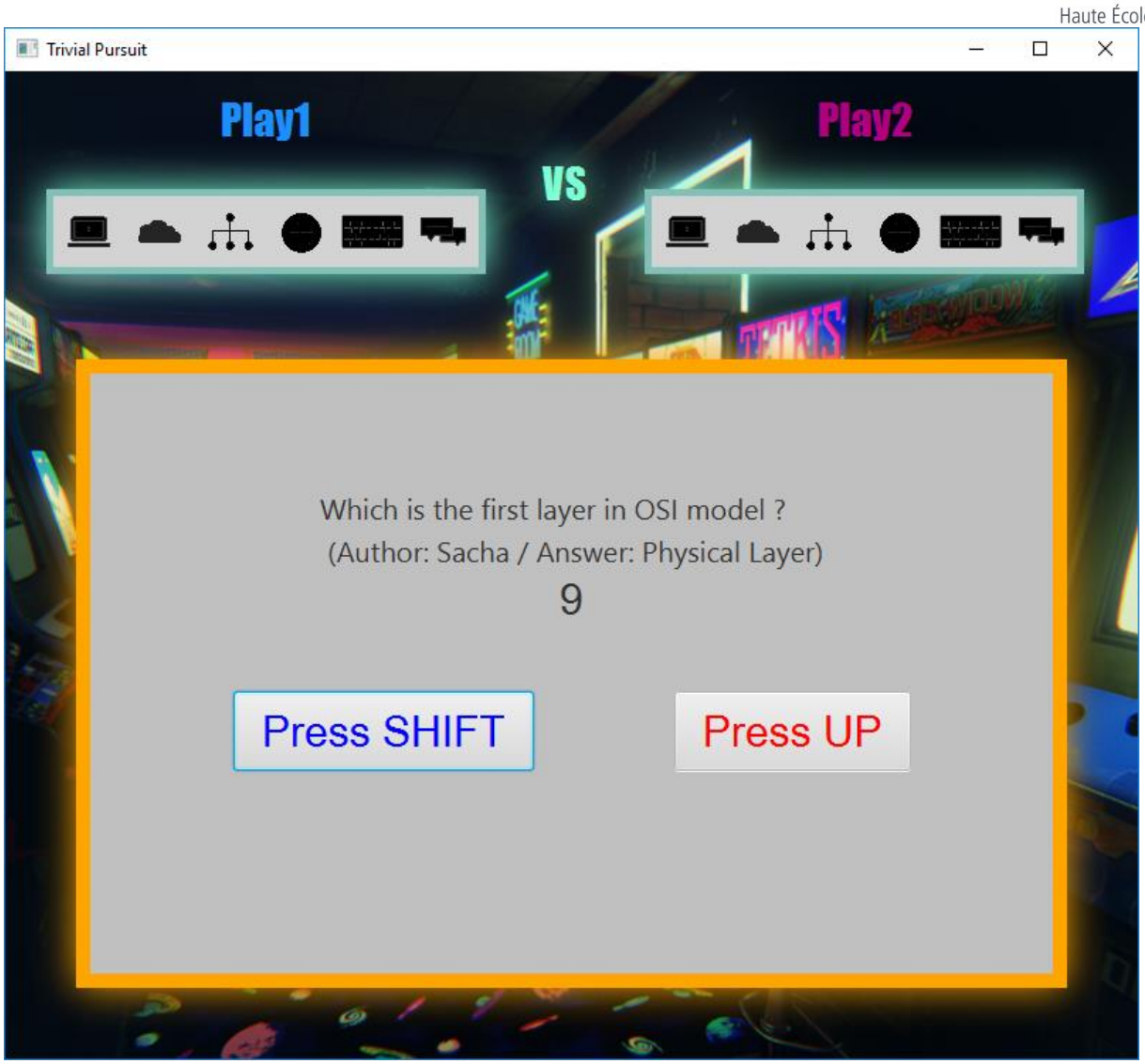
La seule différence du mode solo, c'est que l'on peut ajouter le nom de chaque joueur dans les zones de texte et qu'il n'y a plus le système de manches.

Après avoir choisi le mode multijoueur, vous démarrez le jeu.

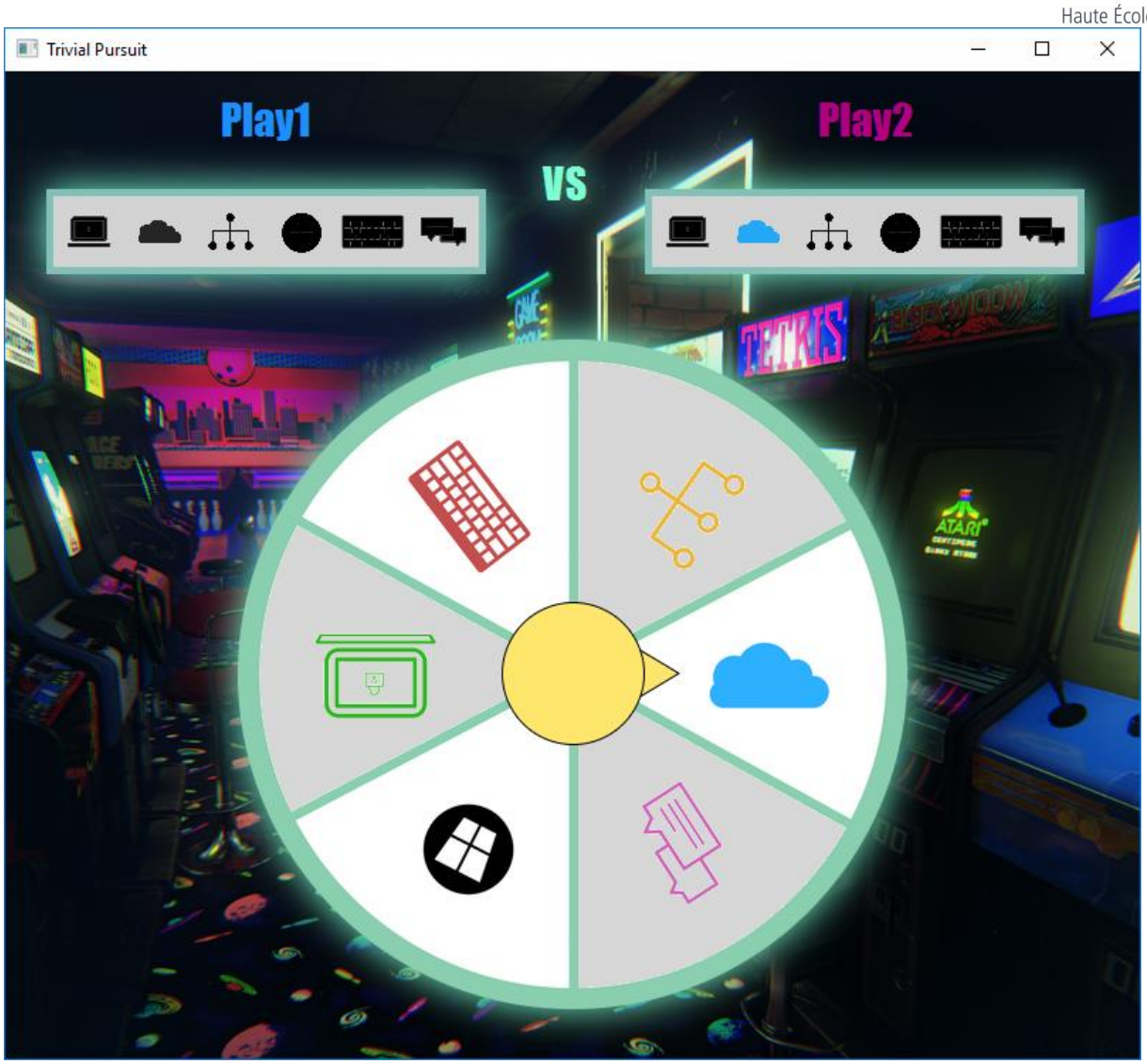


Sur ce mode on retrouvera juste le nom de chaque joueur avec leur barre de camemberts ainsi que notre fameuse roue (sans le bonus et malus rencontrés au mode solo).

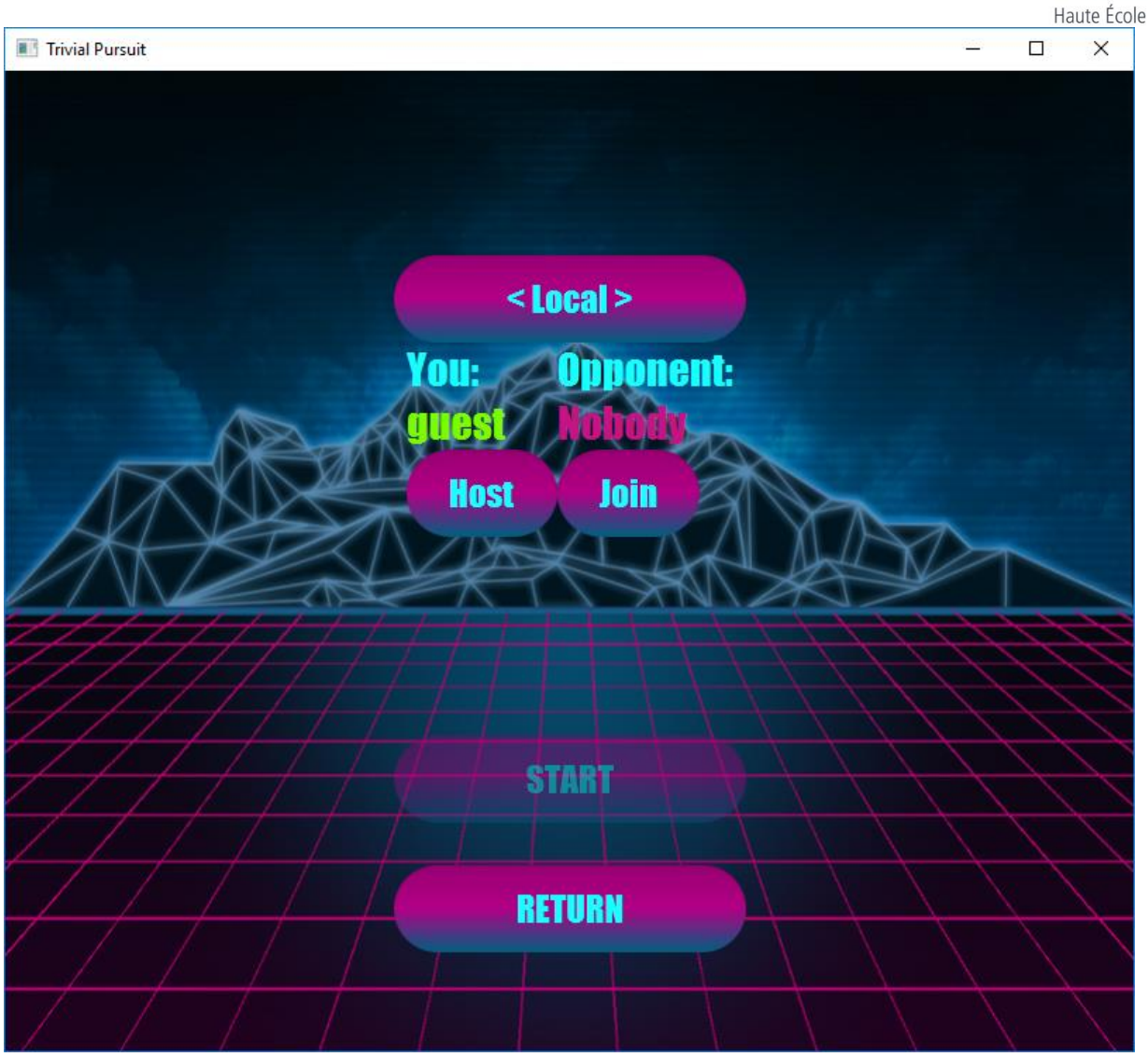
En démarrant la roue, on arrive à la question du thème sélectionné.



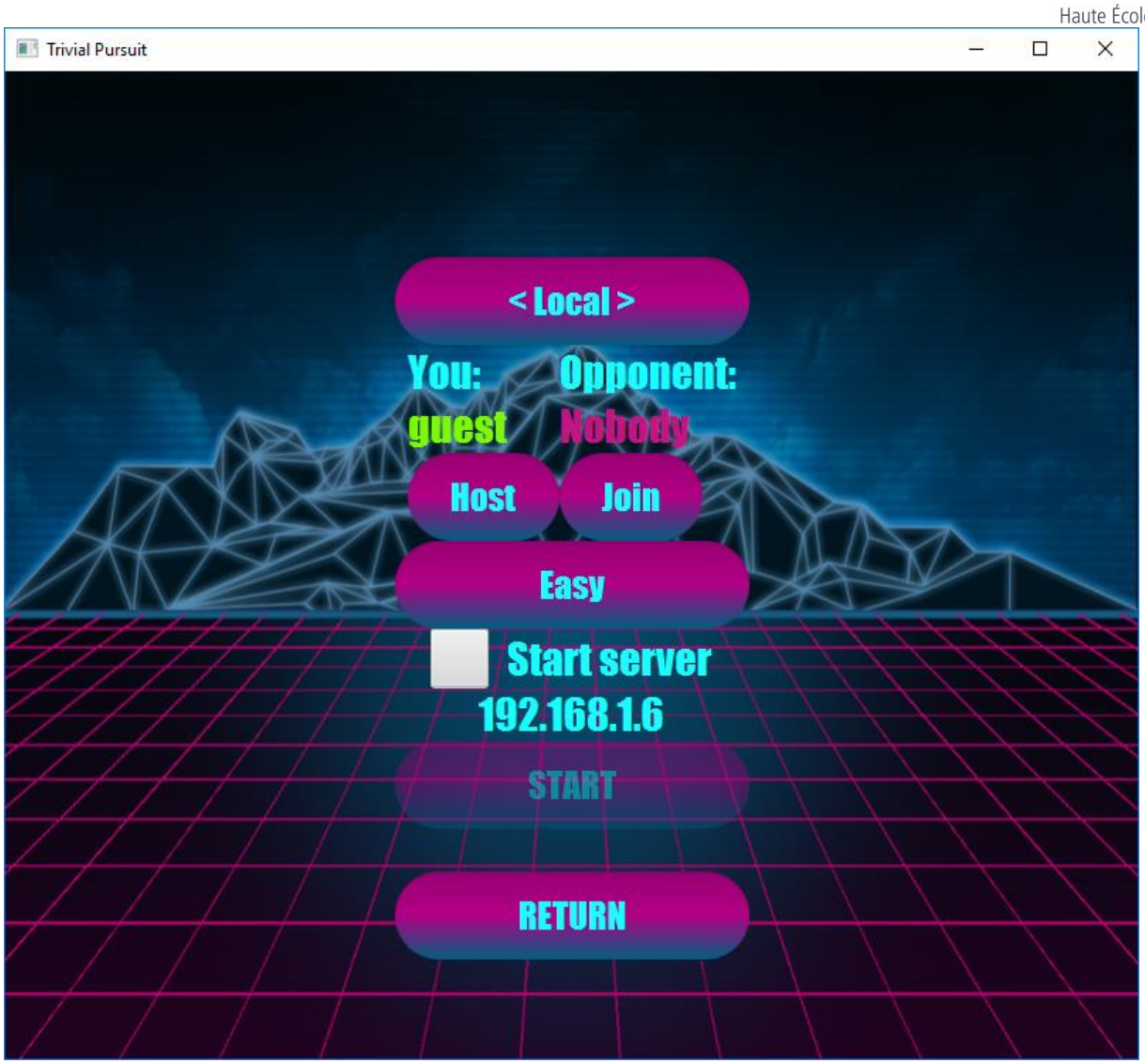
Vous remarquerez que deux boutons se sont ajoutés.
Le joueur le plus rapide qui appuiera sur son bouton, stoppera le délai général de la question et aura un délai de 10 secondes pour répondre à la question et empêchera l'autre joueur de répondre durant ce délai, s'il se trompe le délai global reprendra et l'autre joueur pourra alors tenter sa chance pour recevoir son camembert.



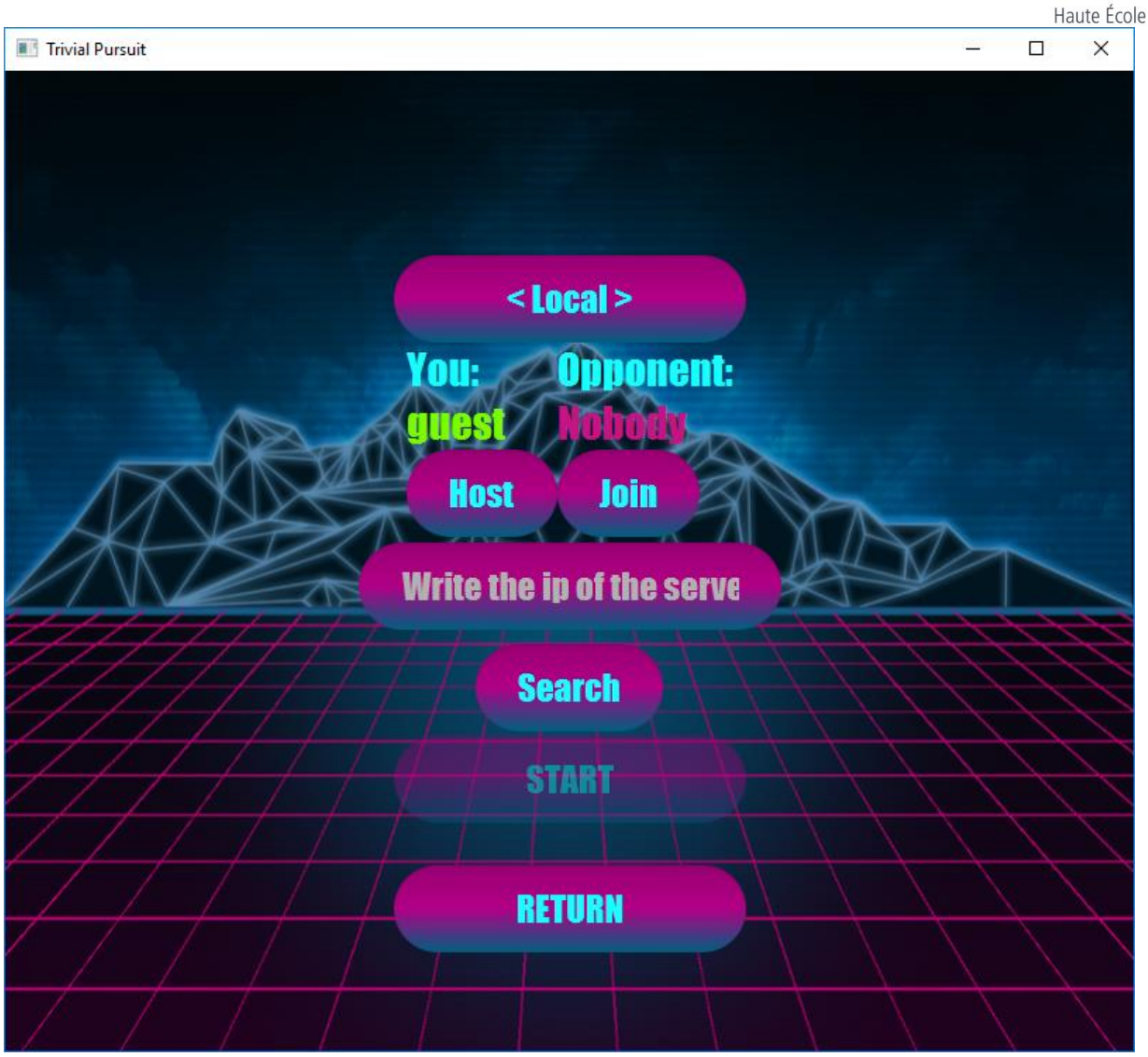
Si vous sélectionnez le mode Local.



Pour jouer en mode local, vous aurez donc deux choix soit héberger ou rejoindre la partie.



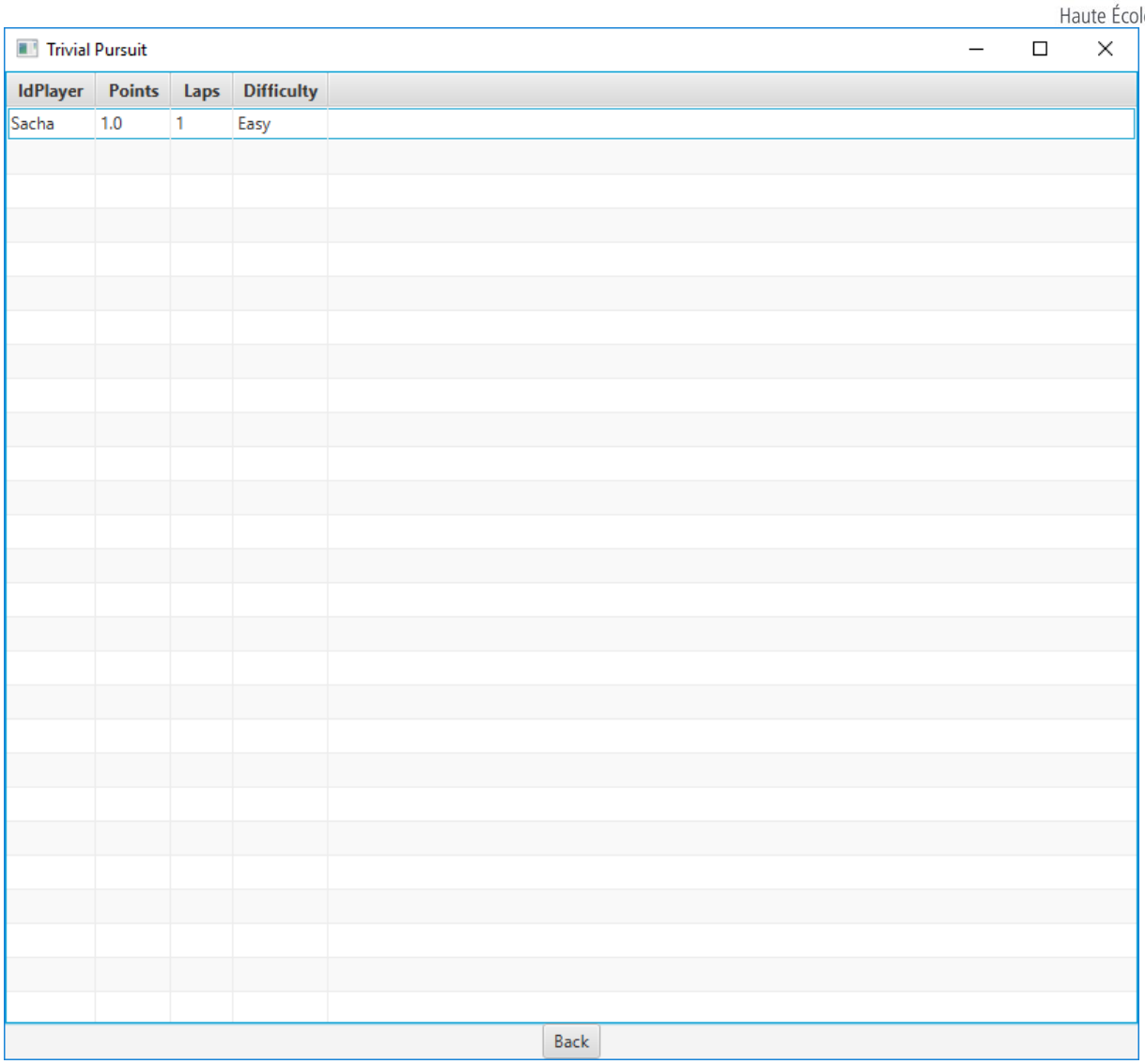
Si vous souhaitez héberger la partie.



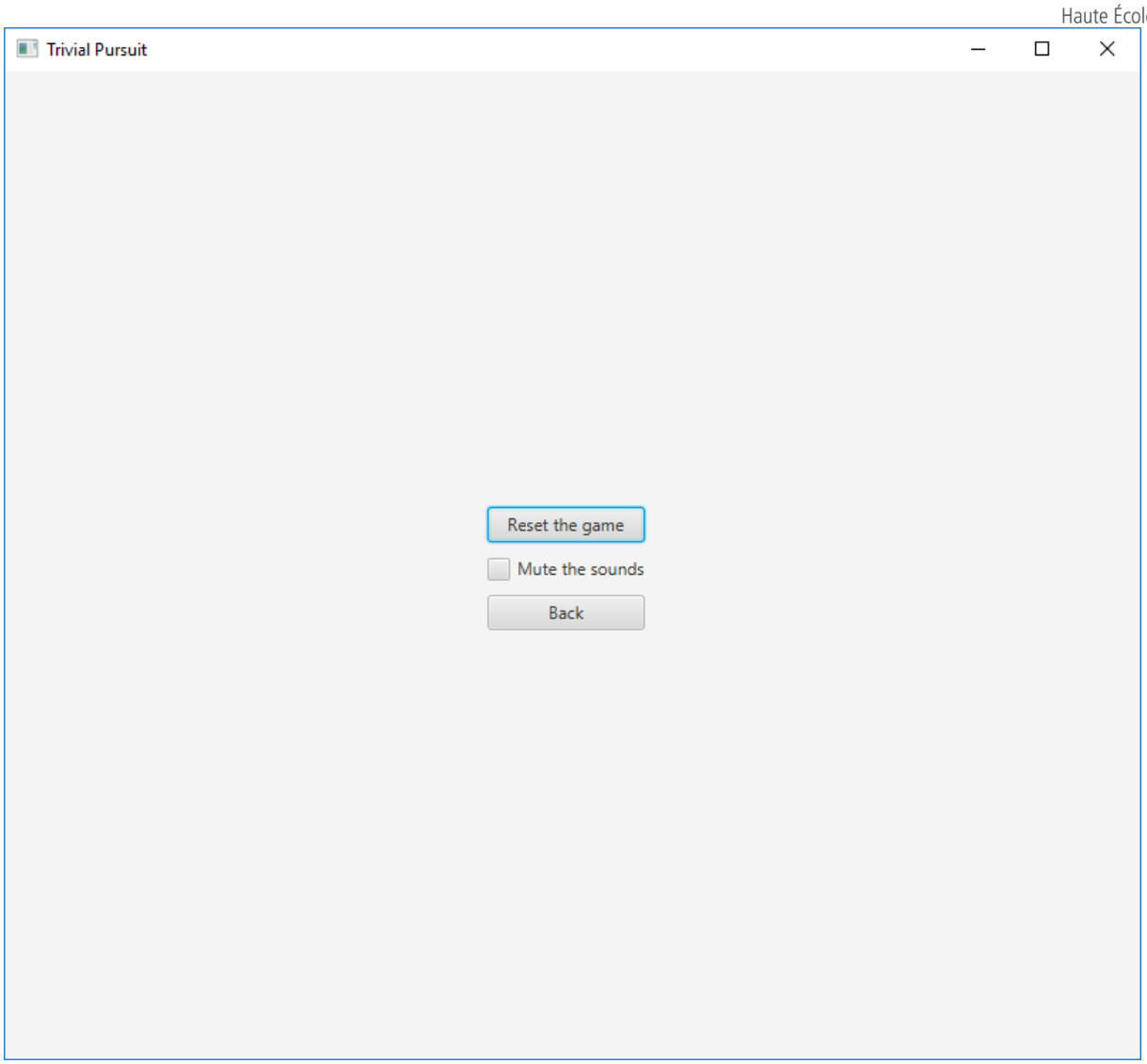
Pour lancer la partie, l'hôte doit donc cocher le start pour lancer le server et donner l'IP au joueur qui souhaite le rejoindre afin qu'il puisse le trouver quand c'est fait les deux joueurs doivent appuyer sur le bouton start.

Le mode local est exactement comme le mode multijoueur à l'exception que le joueur 1 sera toujours le joueur devant l'écran.

Revenons au menu principal.
Si vous cliquez si le bouton "Scoreboard"



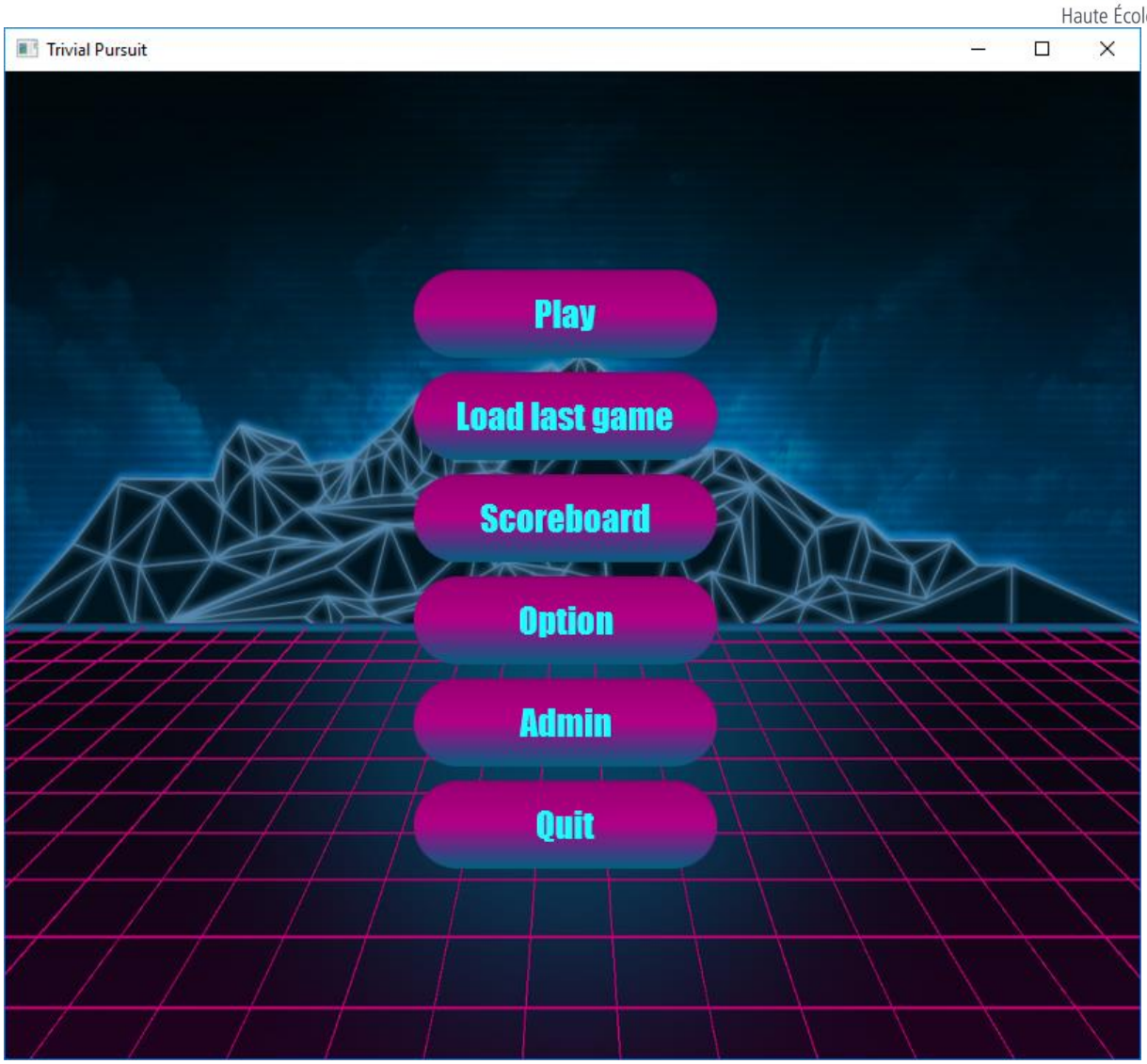
Ça affichera le score des utilisateurs enregistrés avec la difficulté choisie pour leur partie.
Le bouton "Back" vous permet de revenir au menu principal.



De retour au menu principal, si vous cliquez sur le bouton "option".

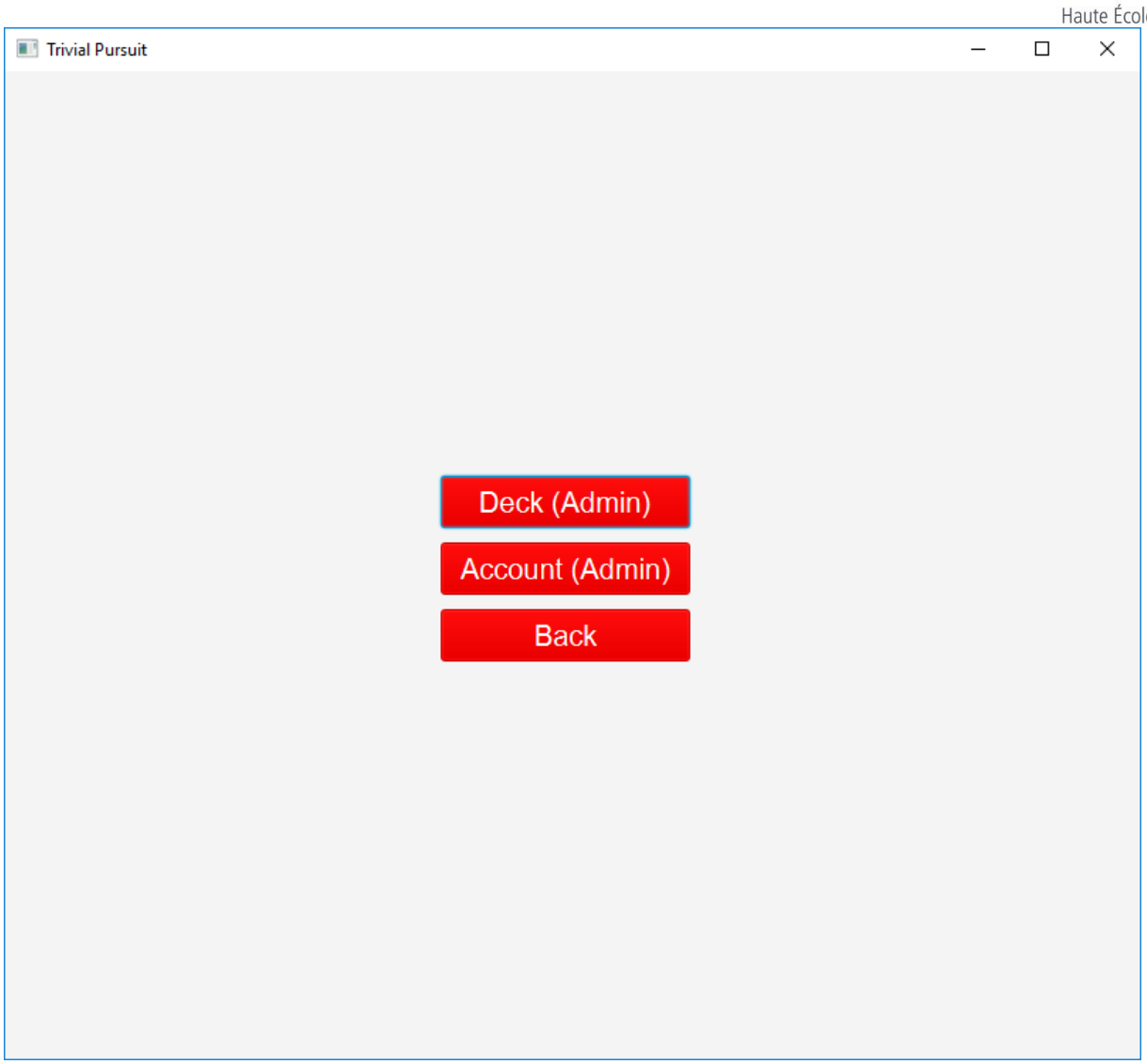
Le bouton "reset the game" vous permet de réinitialiser les questions du deck, les comptes d'utilisateurs et le tableau des scores.
La zone "Mute the sounds" vous permet d'activer ou de désactiver le son du jeu.
Le bouton "Back" vous permet de revenir au menu principal.

Si vous vous connectez avec un compte administrateur, vous aurez un bouton supplémentaire au menu principal.



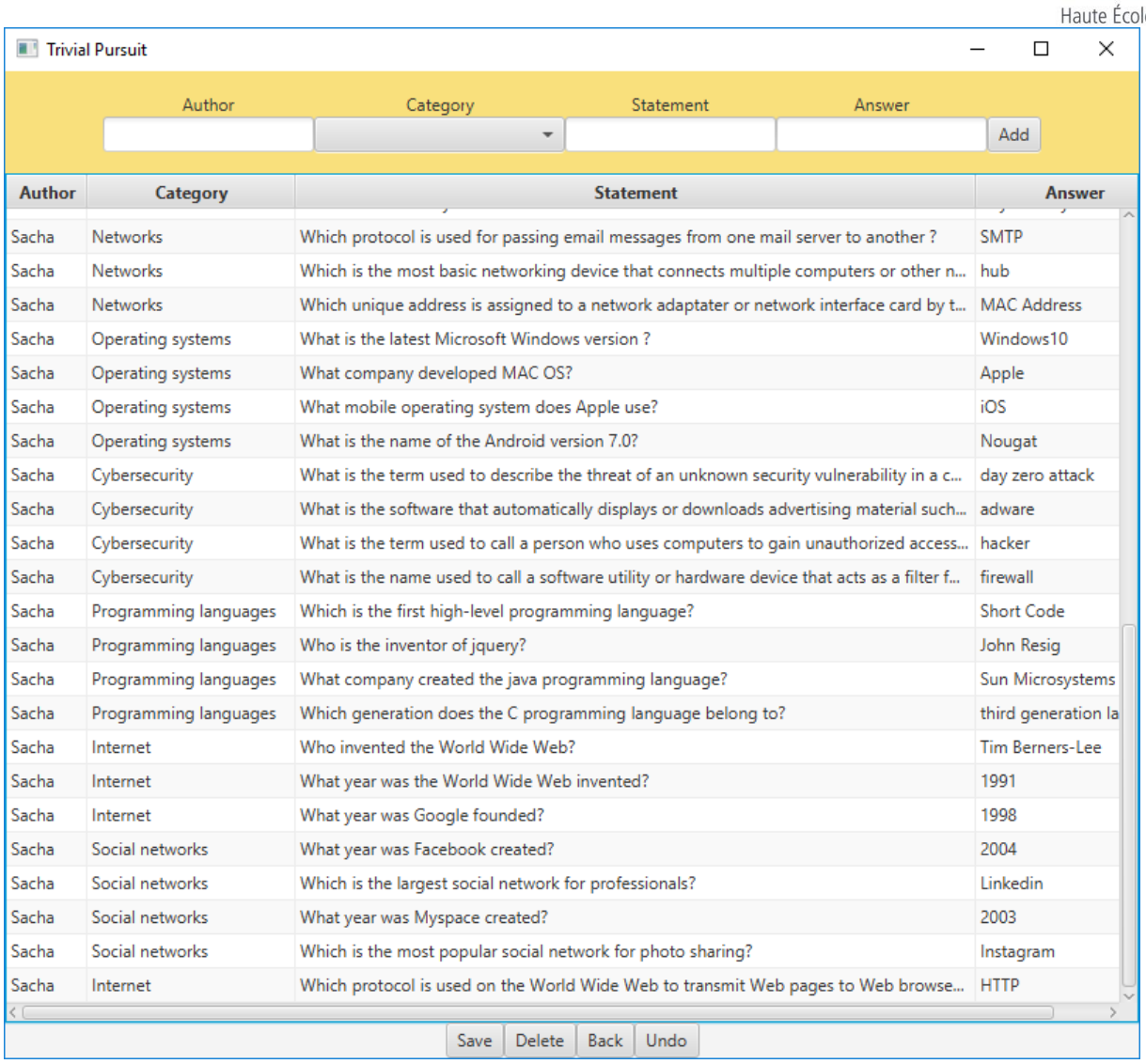
Le bouton "Admin" vous donne accès à des éléments que seul l'admin doit avoir.

Si vous cliquez sur le bouton "Admin".



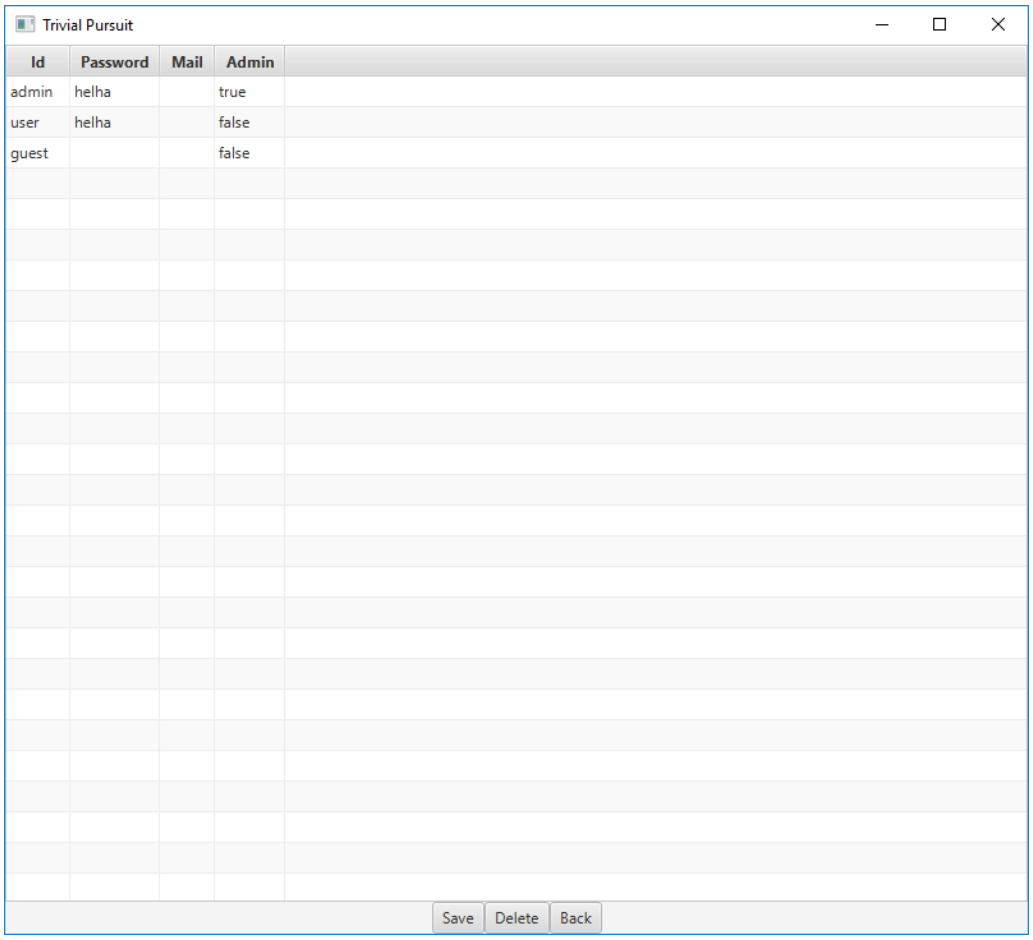
Le bouton "Deck" vous permet d'accéder à toutes les questions du deck.
Le bouton "Account" vous permet de voir tous les comptes inscrits au jeu.
Le bouton "Back" vous permet de revenir au menu principal.

Si vous cliquez sur le bouton "Deck" :



Le zone de texte vous permet d'ajouter une question au thème sélectionné.
Le bouton "Save" vous permet de sauvegarder les modifications.
Le bouton "Delete" vous permet de supprimer une question sélectionnée.
Le bouton "Back" vous permet de revenir l'option admin.
Le bouton "Undo" vous permet d'annuler la dernière modification.

Si vous cliquez sur le bouton "Account" :



Le bouton "Save" vous permet de sauvegarder les modifications.
Le bouton "Delete" vous permet de supprimer un utilisateur.
Le bouton "Back" vous permet de revenir à l'option Admin.

VII. Tests Unitaires

Les tests unitaires nous ont permis de vérifier le bon fonctionnement d'une partie de notre code. Nous en avons donc créé sur les classes les plus importantes telles que les classes Deck, Game et Question.
Voici notre couverture de code pour le package modele :

modele	69,1 %	501	224	725
Player.java	50,0 %	59	59	118
Score.java	21,7 %	15	54	69
ListAccount.java	0,0 %	0	50	50
Account.java	74,5 %	82	28	110
Deck.java	87,1 %	115	17	132
Highscore.java	57,7 %	15	11	26
Game.java	97,8 %	136	3	139
Question.java	97,5 %	79	2	81

Voici un exemple d'un de nos tests :

```
public class DeckTest {

	private Deck deck;
	private List<Question> questions;
	private Question q;

	@Before
	public void setUp() throws Exception {
		deck = deck.getInstanceDeck();
		questions=(List<Question>) Explorateur.getField(deck, "questions");
		q = new Question("autho", Category.CYBERSECURITY, "statement", "answer");
		questions.add(q);
	}

	@After
	public void tearDown() throws Exception {
		questions.clear();
		deck = null;
		questions=null;
		q=null;
	}

	@Test
	public void testGetInstanceDeck() {
		assertEquals("Test de GetInstanceDeck:",Deck.getInstanceDeck(), deck);
	}

	@Test
	public void testAddQuestion() {
		Question q = new Question("autho2", Category.CYBERSECURITY, "statement2",
"answer2");
		deck.addQuestion(q);assertEquals("Test de addQuestion()", deck.getQues-
tions().get(1), q);
		assertEquals("Test de AddQuestion()", deck.getQuestions().size(), 2);
	}

	@Test
	public void testGetQuestions() {
		assertEquals("Test de GetListQuestion()",deck.getQuestions(), questions);
	}

	@Test
	public void testSetQuestions() {
		Question q=new Question("author3", Category.INTERNET, "statement3", "answer3");
		List listQuestion = new ArrayList<>();
		listQuestion.add(q);
		deck.setQuestions(listQuestion);
		assertEquals("Test de SetListQuestion()",deck.getQuestions(),listQuestion);
	}

	@Test
	public void testToDraw() {
		Category categ = Category.CYBERSECURITY;
		Question quest = deck.toDraw(categ);
		assertEquals("Test de ToDraw()",quest.getCategory(),categ);
	}}
```

VIII. Conduite de Projet

Sprint 0		
US	Tasks	Author
US-01	Créér les classes	Loïc / Sacha
US-02	Afficher les questions	Loïc
US-03	Répondre à une question	Loïc

Sprint 1		
US	Tasks	Author
US-01 En tant qu'utilisateur je peux lancer une partie en mode express (solo)	Créér la roue	Sacha
	Lancer la roue choisit une catégorie et une question aléatoire	Loïc
	La question s'affiche et un champ est disponible pour répondre	Loïc
	Un timer valide le champ lorsqu'il atteint 0	Loïc
	Mise en place d'un système de calculs de points pour le score	Loïc
	Ajout d'une portion par bonne réponse pour la catégorie en question	Loïc
US-02 En tant qu'administrateur je peux gérer un deck	Ajouter une question	Loïc
	Supprimer une question	Loïc
	Modifier une question	Loïc
	Gérer les doublons lors de l'ajout	Loïc

Sprint 2		
US	Tasks	Author
US-01 En tant qu'utilisateur je peux ouvrir un menu d'option en jeu	Le bouton Save permet de sauvegarder la partie	Bilal
	Le bouton Exit permet de quitter la partie	Bilal
	Le bouton Return permet de revenir au menu principal	Bilal
	Le bouton Cancel permet de fermer la fenêtre	Bilal
US-02 En tant qu'utilisateur je peux jouer une partie en mode duel	Lancer la roue choisit une catégorie et une question aléatoire	Sacha
	Affichage d'une question pour les deux joueurs	Sacha
	Mise en place d'un système de buzzer	Sacha
	Mise en place d'un timer général (pour la question)	Sacha
	Mise en place d'un timer pour chaque joueur	Sacha
	Ajout d'une portion pour le joueur ayant bien répondu à la question de la catégorie	Sacha
US-03 En tant qu'utilisateur je peux me connecter	Se connecter en tant qu'administrateur	Loïc/Sacha
	Se connecter en tant qu'utilisateur classique	Sacha
US-04 En tant qu'utilisateur je peux sauvegarder et reprendre une partie	Sauvegarder une partie en mode Solo	Loïc
	Reprendre une partie en mode Solo	Loïc
	Sauvegarder une partie en mode Duel	Loïc
	Reprendre une partie en mode Duel	Loïc

Sprint 3		
US	Tasks	Author
US-01 En tant qu'utilisateur je peux avoir un compte personnel	Connexion en tant qu'invité	Loïc
	Création d'un compte	Loïc
	Envoie d'un mail après inscription	Loïc
	Accès d'un panneau administrateur si connecté en tant qu'administrateur	Loïc
	Sauvegarde liée au compte	Loïc
US-02 En tant qu'administrateur je peux gérer les comptes	Ajouter un compte	Loïc
	Modifier un compte	Loïc
	Supprimer un compte	Loïc
US-03 En tant qu'utilisateur je peux jouer en multijoueur en réseau local	Héberger une partie	Loïc/Bilal
	Rejoindre une partie via l'adresse IP	Loïc/Bilal
	Même principe de jeu que pour le mode Duel	Loïc/Bilal
US-04 En tant qu'utilisateur je peux accèder à un menu pour configurer ma partie	Choix du mode de Jeu	Sacha
	Paramétrer certaines variables de la partie	Sacha
	Lancer la partie	Sacha
US-05 En tant qu'utilisateur je veux avoir une belle interface	Interface plus colorée et uniforme	Sacha/Loïc
	Question entourée de la couleur de la catégorie	Sacha/Loïc
US-06 En tant qu'utilisateur je peux voir le résultat de ma réponse	Une alerte s'ouvre en fonction du résultat	Bilal
	Un son est joué en fonction du résultat	Bilal

IX. Conclusion

A. Les difficultés rencontrées

Le problème le plus important rencontré fut les timers qui se gênaient mutuellement, ou qu'ils ne s'arrêtaient pas forcément. Mais cela nous a permis de faire de plus amples recherches pour trouver la solution à ce problème. Nous avons pu régler ce problème en ajoutant plusieurs solutions.
La mise en place du réseau fut aussi assez compliquée car c'est une nouvelle matière que ne connaissions pas. Il a fallu un grand temps d'adaptation et de compréhension pour qu'uniquement deux joueurs puissent se connecter l'un à l'autre mais finalement, une fois cette difficulté surmontée, la mise en place du système de jeu fut beaucoup plus facile.

B. Ce que le projet nous a apporté

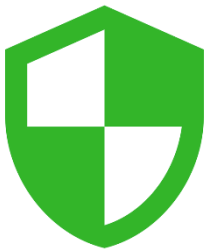
Pour conclure, nous pouvons dire que ce projet a été un entrainement très efficace dans plusieurs domaines. Ce projet nous a permis, premièrement, d'approfondir les connaissances que nous avons vu en cours car nous avons travaillé de manière autonome pour découvrir et enrichir nos connaissances. Nous avons dû pour cela, faire preuve de patience et effectuer de nombreuses recherches, ainsi qu'une mise en commun de celles-ci, pour former des échanges bénéfiquement mutuels. Et dans un second temps, d'améliorer notre travail d'équipe ainsi qu'une meilleure organisation et gestion, pour ce type d'exercice. En plus de cela, nous avons surpassé nos attentes. Nous avons fait beaucoup plus que ce que nous nous pensions être capable de faire, comme réalisé un mode de jeu en réseau ou bien encore d'utiliser des notions plus « compliqués » comme l'introspection ou les threads.

Annexes

#Bug1 : événement indésirable, le timer ne s'arrêtait pas lorsque les deux joueurs avaient une mauvaise réponse pour le mode Duel. La correction a été effectuée dans la classe QuestionDualPane dans la méthode confirmAnswer

```
if(response != null) {
                    if(response == false ) {
                          util.Utility.getMediaPlayer("src/son/sonF.mp3").play();
                          MainFx.getStackGamePane().getAlertInformation("You are bad
!").show();
                          if(alreadyBuzz1 == false || alreadyBuzz2 == false) {
                                startTimelineQuestion();
                          }
                    }
                    if(response == true) {
                          util.Utility.getMediaPlayer("src/son/sonT.mp3").play();
                          MainFx.getStackGamePane().getAlertInformation("Congratulations
!").show();
                    }
              }
```

#Modifications : une image pour la portion de roue, pour une question de visibilité (l'icône pour la cybersécurité)
La nouvelle image / Ancienne image



Modification d'une partie du code : cette méthode permet de réinitialiser les fichiers « json » mais ne fonctionnait pour le fichier « deck », elle a donc été réécrite

```
public static void reset(String file, String file2, Class c) {
Object obj = loadFile(c, file);
saveFile(file2, obj);
 }
```