Fiche de Synthèse : Urban Marginal *Avril 2020*

Développement :
Urban Marginal est un jeu de combat en 2D, client-serveur : les joueurs doivent toucher l'adversaire avec leur boule, jusqu'à ce que celui-ci n'ait plus de vies.

Cadre :
Il s'agit d'un TP proposé dans le cadre du cours " Programmation Objet ", par Elisabeth Martins Da Silva.
Les images nous étaient fournies, ainsi que les instructions pas à pas pour le développement.

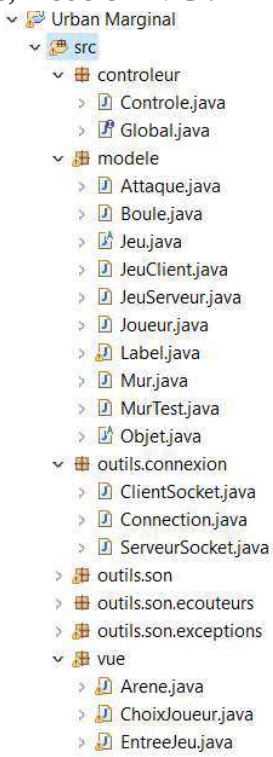
Support :
Java, à l'aide d'Eclipse Néon incluant le plugin WindowBuilder.

Contraintes / Organisation :
Ce jeu a été développé selon le modèle MVC : Modèle – Vue – Contrôleur.
Le package Modèle contient les classes métiers, la Vue les différentes fenêtres du jeu, et le Contrôleur la classe qui va gérer les intéractions entre toutes ces classes.
Un package " Outils " a été ajouté, qui contient toutes les classes techniques (pour la connection et le son)

Difficultés rencontrées :
Il est difficile, en cas d'erreur, de trouver où celle-ci se trouve. J'ai passé…….un temps infini, plusieurs heures, à rechercher une erreur de chargement d'image, sans comprendre d'où elle venait. C'était une erreur dans mes variables globales…...un " s " sur le chemin d'accès aux murs qui n'aurait pas dû être là.
Du moins cette mésaventure m'a permi de comprendre un peu mieux le fonctionnement d'Eclipse (et ses exceptions). La prochaine fois que je rencontrerai ce type d'erreur, je passerai moins temps à l'identifier et la corriger.

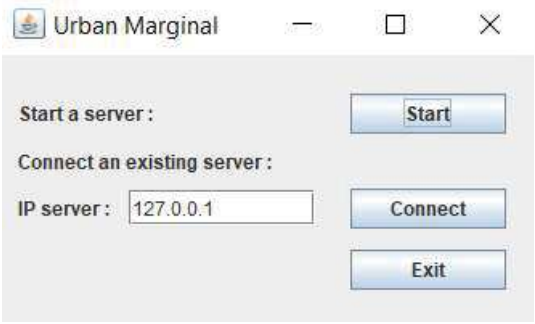
Description résumée du développement

1. Généralités:
Voici donc les classes qui ont été créées, modèle MVC :

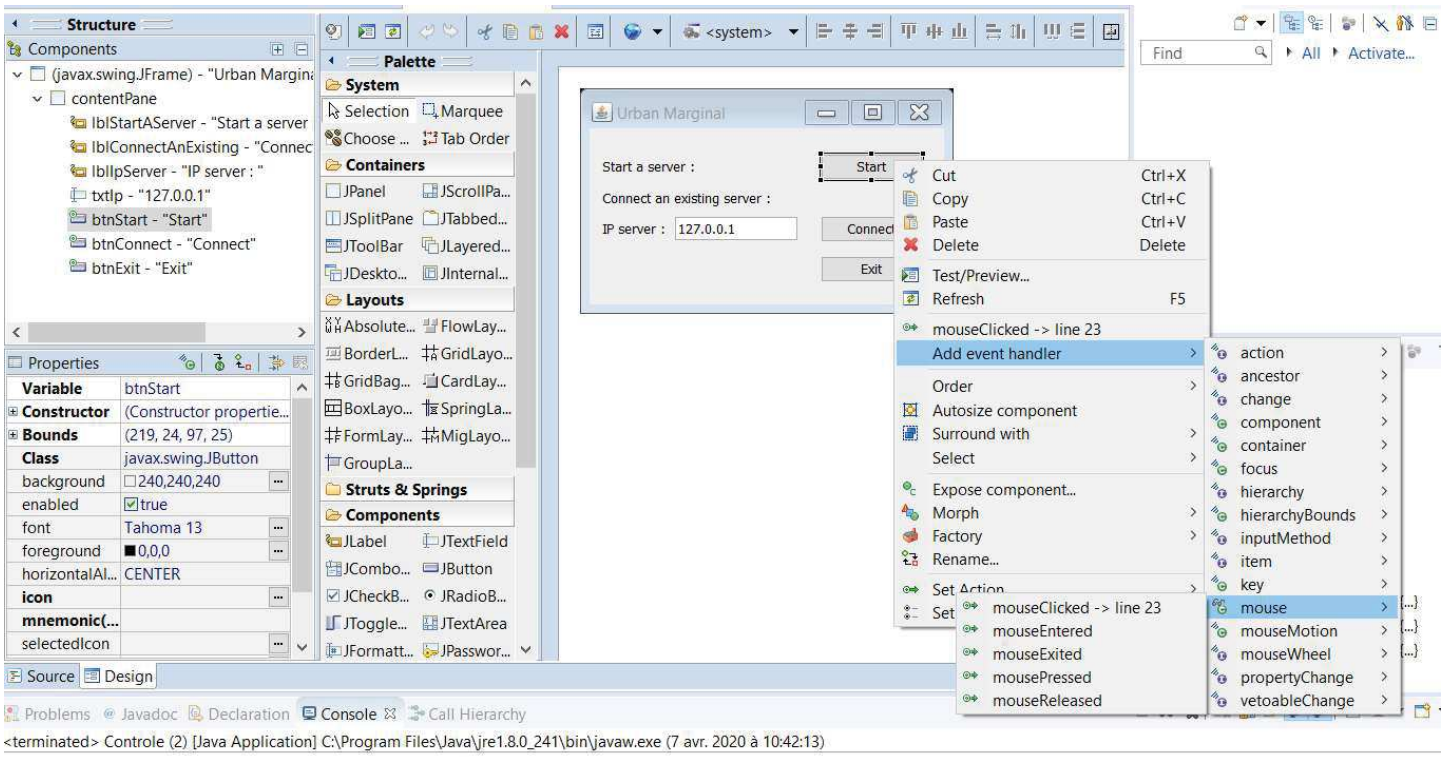


Les classes techniques relatives aux sons du jeu nous ont été fournies.

2. Fenêtre n°1 : l'entrée dans le jeu :



L'utilisateur doit, à ce moment, choisir s'il démarre un serveur (dans ce cas il ne joue pas), ou s'il se connecte à un serveur déjà lancé.
La fenêtre a été dessinnée grâce à WindowBuilder : il suffit de choisir et positionner les élements, le code correspondant est automatiquement généré. C'est ainsi que la classe EntreeJeu est créée.
On peut ensuite ajouter les évènements que l'on souhaite :



La classe Controle est créée dès ce moment là, pour instancier une EntreeJeu.

Puis toutes les classes relatives aux connections : la classe ServeurSocket, qui créé un socket pour que le serveur soit à l'écoute des clients ; la classe Connection, qui hérite de Thread, dans laquelle sera créé le canal d'entrée et le canal de sortie ; et enfin la classe ClientSocket qui va se connecter au serveur.
Dans ces classes techniques, le try/catch est indispensable : il est important, en cas d'échec de connexion, de savoir où et pourquoi.
Un exemple de la méthode run() de la classe ServeurSocket :

```
/**
 * Thread qui va attendre la connexion d'un client
 */
public void run() {
    Socket socket;
    // boucle infinie pour attendre un nouveau client
    while (true) {
        try {
            System.out.println("le serveur attend");
            socket = serverSocket.accept();
            System.out.println("un client s'est connecté");
            new Connection(socket,leRecepteur);
        } catch (IOException e) {
            System.out.println("impossible de récupérer le socket du client ! : " + e);
            System.exit(0);
        }
    }
}
```

Ainsi les affichages relatifs aux connections s'affichent au fur et à mesure dans la console d'Eclipse, ce qui nous evitera bien des soucis par la suite.

Dans le Modèle, la classe mère abstraite Jeu est créée, puis ses filles JeuClient et JeuServeur qui redéfinissent ses méthodes.

Ex JeuServeur :

Constructor Summary

Constructors

Constructor	Description
`JeuServeur(Controle controle)`	Constructeur

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
void	`constructionMurs()`	Méthode qui va s'occuper de générer les murs
void	`deconnection(Connection connection)`	Pour gérer la déconnection à l'ordinateur distant
void	`envoi(java.lang.Object info)`	Envoi à tous les clients
void	`nouveauLabelJeu(Label label)`	Demande au controleur d'ajouter un joueur dans l'arène
void	`reception(Connection connection, java.lang.Object info)`	Traitement de la réception de messages provenant de l'ordinateur distant
void	`setConnection(Connection connection)`	Gestion de la réception d'un objet Connection

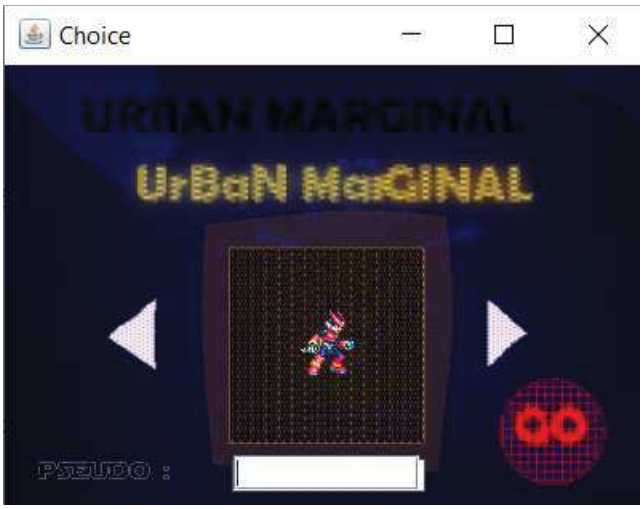
A ce stade évidemment, outre le constructeur, seules les méthodes deconnection(), reception() et setConnection() sont crées. Les mêmes côté JeuClient
Le Controle, comme il se doit, gèrera les connections et déconnections : il est capable, magie de l'héritage, de savoir si c'est un client ou un serveur qui se connecte.

```
/**
 * Récupération de la connexion
 * @param connection
 */
public void setConnection(Connection connection) {
    this.connection = connection;
    if (leJeu instanceof JeuServeur) {
        leJeu.setConnection(connection);
    }
}

public void deconnection(Connection connection) {
    leJeu.deconnection(connection);
}
```

N'oublions pas, pour la suite du développement, la classe Global : comme son nom l'indique, elle regroupera toutes les variables globales qui nous serviront pour la suite :

```
package controleur;

public interface Global {

    public static final int PORT = 6666;

    // fichiers :
    public static final String SEPARATOR = "//";
    public static final String CHEMIN = "media" + SEPARATOR;
    public static final String CHEMINFONDS = CHEMIN + "fonds" + SEPARATOR;
    public static final String CHEMINPERSOS = CHEMIN + "personnages" + SEPARATOR;
    public static final String CHEMINMURS = CHEMIN + "murs" + SEPARATOR;
    public static final String PERSO = CHEMINPERSOS + "perso";
    public static final String EXTIMAGE = ".gif";
    public static final String CHEMINBOULES = CHEMIN + "boules" + SEPARATOR;
    public static final String CHEMINSONS = CHEMIN + "sons/";

    // images :
    public static final String FONDCHOIX = CHEMINFONDS + "fondchoix.jpg";
    public static final String FONDARENE = CHEMINFONDS + "fondarene.jpg";
    public static final String MUR = CHEMINMURS + "mur.gif";
    public static final String BOULE = CHEMINBOULES + "boule.gif";

    // personnages :
    public static final int GAUCHE = 0;
```

2. Fenêtre 2 : Le choix du Joueur



Accessible uniquement par le client, cette fenêtre a, en plus de la JtextFiled (la zone de saisie du pseudo) des Jlabel qui ont été placés sur les flèches et le bouton Go. Ainsi, l'évènement sera sur le clic du Jlabel correspondant.
Par exemple, sur le clic du bouton Go, la méthode lblGo_clic() sera appelée :

```
JLabel lblGo = new JLabel("");
lblGo.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        lblGo_clic();
    }
    @Override
    public void mouseEntered(MouseEvent e) {
        souris_doigt();
    }
    @Override
    public void mouseExited(MouseEvent e) {
        souris_normale();
    }
});
lblGo.setBounds(309, 196, 68, 67);
contentPane.add(lblGo);
```

Méthode écrite au-dessus du constructeur de la fenêtre, pour une meilleure lisibilité du code :

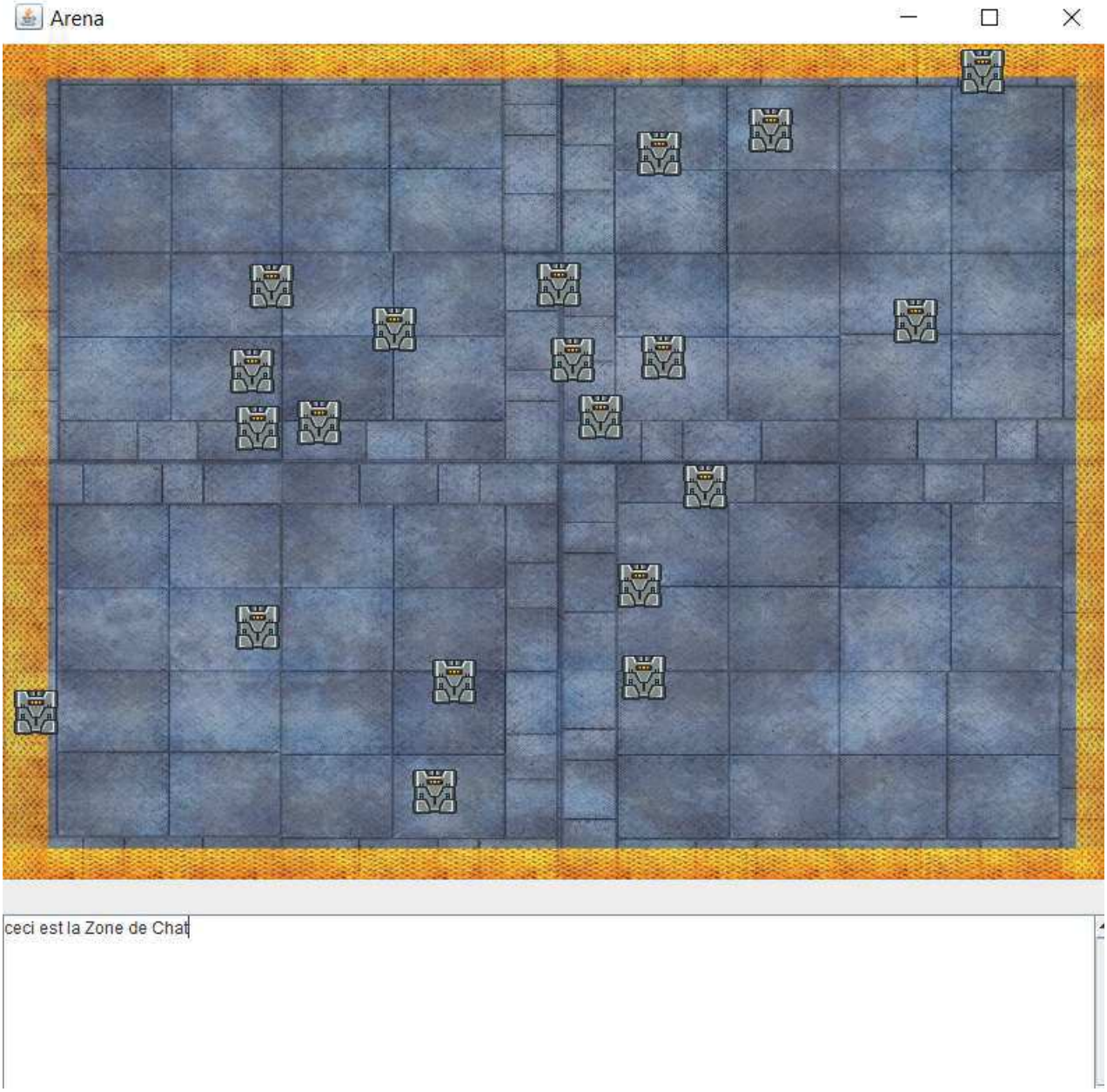
```
/**
 * Clic sur "Go"
 */
private void lblGo_clic() {
    if(txtPseudo.getText().equals("")) {
        JOptionPane.showMessageDialog(null, "le pseudo est obligatoire !");
        txtPseudo.requestFocus();
    } else {
        go.play();
        controle.evenementVue(this, PSEUDO + SEPARE + txtPseudo.getText() + SEPARE + numPerso);
    }
}
```

Une fois les évènement créés sur les boutons précédent et suivant, ainsi que le choix du personnage et le pseudo, il est temps d'indiquer au controleur comment gérer tout ça :

```
/**
 * gère les évènements provenant de la frame ChoixJoueur
 * @param info
 */
private void evenementChoixJoueur(Object info) {
    ((JeuClient)leJeu).envoi(info);
    frmChoixJoueur.dispose();
    frmArene.setVisible(true);
}
```

Oups, le controleur lance l'arène, il est temps qu'on passe à son développement.

3. L'arène



Avec ses murs qui sont générés à des endroits aléatoires, une zone de Chat pour que les joueurs puissent discuter, l'endroit où le personnage du joueur, avec son pseudo et sa vie, apparaîtront.

C'est le serveur qui va gérer tous les calculs, et les clients vont se contenter d'afficher ce que le serveur leur envoi.

Les joueurs, les murs, les boules etc....seront des Jlabel : pour pouvoir les identifier (il faut qu'on puisse savoir s'il s'agit d'un nouveau (un joueur qui se connecte) ou d'un label existant qu'il faut déplacer) il va falloir leur attribuer un numéro.

Une classe Label est donc créée : c'est en quelque sorte un Jlabel à qui on a donné une carte d'identité :

Constructor	Description
`Label(int numLabel, javax.swing.JLabel jLabel)`	Constructeur

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`javax.swing.JLabel`	`getjLabel()`	Getter
`static java.lang.Integer`	`getNbLabel()`	Getter
`java.lang.Integer`	`getnumLabel()`	Getter
`static void`	`setNbLabel(int nbLabel)`	Setter

Ensuite, la classe Objet, qui sera la mère…..non pas de toute les classes comme Object, mais la mère des objets Joueur, Mur et Boule. Ils ont donc une position, qu'on peut récupérer ou modifier, et une méthode toucheObjet() est créée : en effet, les joueurs, ni leurs boules d'ailleurs, ne doivent traverser les murs, ou eux-même : il faut donner l'illusion de solidité.

Constructors

Constructor	Description
`Objet()`	

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`Label`	`getlabel()`	Getter
`int`	`getposX()`	Getter
`int`	`getposY()`	Getter
`void`	`setPosX(int posX)`	
`void`	`setPosY(int posY)`	
`boolean`	`toucheObjet(Objet objet)`	contrôle si l'objet actuel touche l'objet passé en paramètres

4. Les combats

C’est à ce moment qu’il faut développer la classe Attaque, qui hérite de Thread. Elle s’intéresse à ce qui se passe quand une boule est tirée, si la victime est touchée ou non, si la victime est morte ou non.

Sa méthode run() pour illustrer :

```
    /**
         Boule laBoule = (this.attaquant.getBoule());
         int orientation = this.attaquant.getOrientation();
         laBoule.getlabel().getjLabel().setVisible(true);
         Joueur victime = null;
         do {
             if (orientation == GAUCHE) {
                 laBoule.setPosX(laBoule.getposX() - LEPAS);
             } else {
                 laBoule.setPosX(laBoule.getposX() + LEPAS);
             }
             laBoule.getlabel().getjLabel().setBounds(laBoule.getposX(), laBoule.getposY(),
L_BOULE, H_BOULE);
             pause(10,0);
             this.jeuServeur.envoi(laBoule.getlabel());
             victime = toucheJoueur();
         } while (laBoule.getposX() >= 0 && laBoule.getposX() <= L_ARENE && !toucheMur() &&
victime == null);
         if (victime != null && !victime.estMort()) {
             jeuServeur.envoi(HURT);
             victime.perteVie();
             attaquant.gainVie();
             for (int i = 1; i <= NBETATSBLESSE; i++) {
                 victime.affiche(BLESSE, i);
                 pause(80,0);
             }
             if (victime.estMort()) {
                 jeuServeur.envoi(DEATH);
                 for (int i = 1; i <= NBETATSMORT; i++) {
                     victime.affiche(MORT, i);
                     pause(80,0);
                 }
             } else {
                 victime.affiche(MARCHE, 1);
             }
         }

         attaquant.affiche(MARCHE, 1);
         laBoule.getlabel().getjLabel().setVisible(false);
         this.jeuServeur.envoi(laBoule.getlabel());
     }
```

Et c’est ainsi que Marguerite tua Gérard...