


Projet


Classement

Réglage

Choix de classe =



Point de vie : 60
Points de mouvements : 2
Distance des attaques : 1
Points d'action : 2



Point de vie : 50
Points de mouvements : 1
Distance des attaques : 2
Points d'action : 2

Votre nom de combattant =

Alexandre

Valider

Projet

Attaque

Compétence

Déplacement

Soin

45 / 50

PV

Guerrier

Alexandre VS Vague 3

Points de mouvement : 1

Points d'action : 2

Projet

Chargement :

- Le joueur choisi une classe (Guerrier ou Archer)
- Le joueur choisi un pseudo
- Début de la partie

Début de la manche :

- Un ennemie et le joueur apparaît dans les 2 parties du terrain de jeu (à côté zone orange)
- Les points de vie, points de mouvement, compétence et points d'action sont initialisés en fonction de la classe et de la manche précédente
- Le nombre d'ennemis en fonction du niveau de vague
- Un coffre contenant une potion peut aussi apparaître

En cours de la manche :

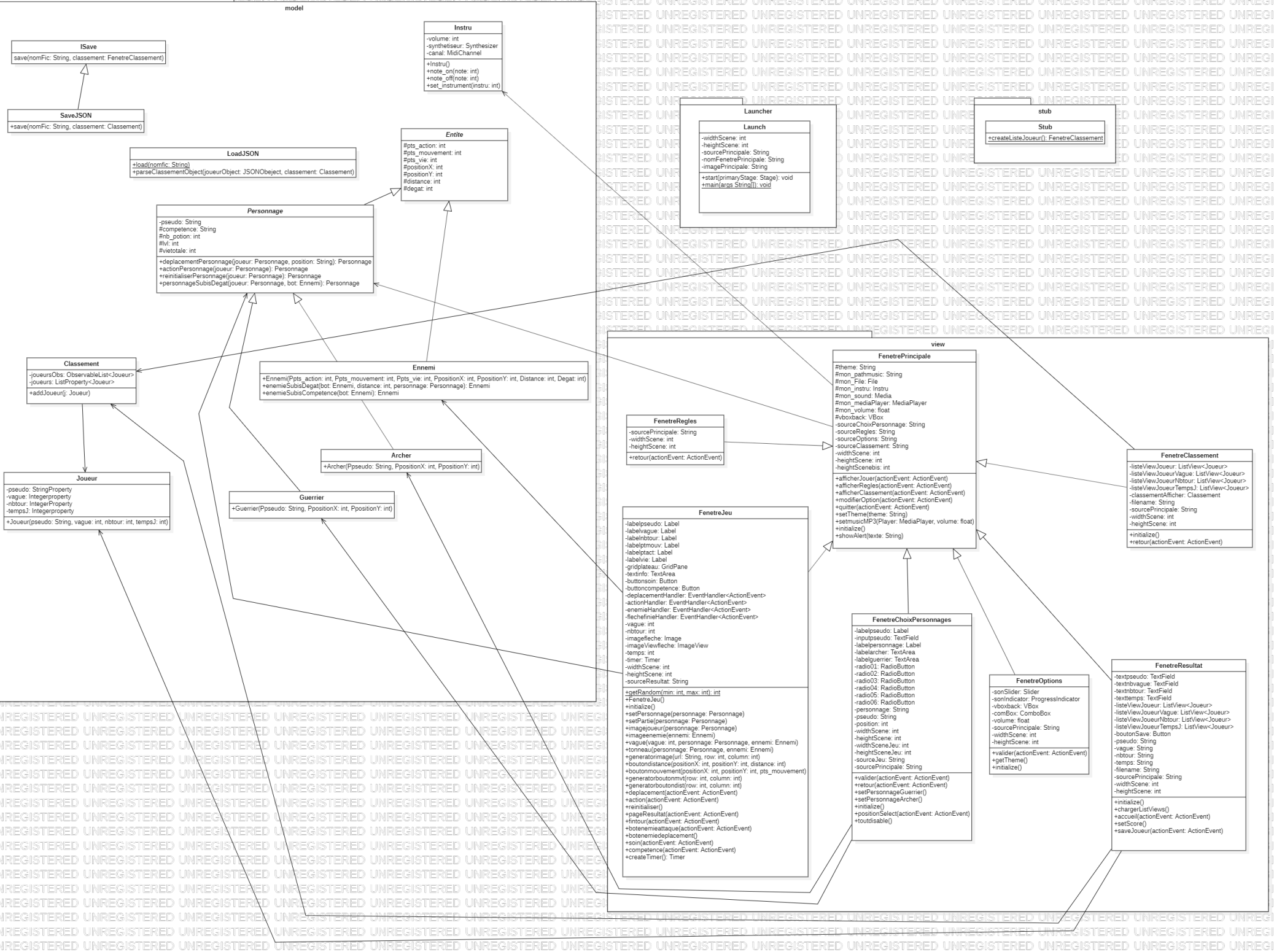
- L'ennemi bouge en fonction de ses points de mouvement aléatoirement (ou intelligemment vers le joueur) et peut attaquer en fonction du type d'ennemi et la proximité du joueur
- Le joueur peut bouger en fonction de ses points de mouvement et peut attaquer en fonction de la classe et la proximité de l'ennemi, il est aussi possible d'ouvrir un coffre, prendre un soin ou utiliser une compétence

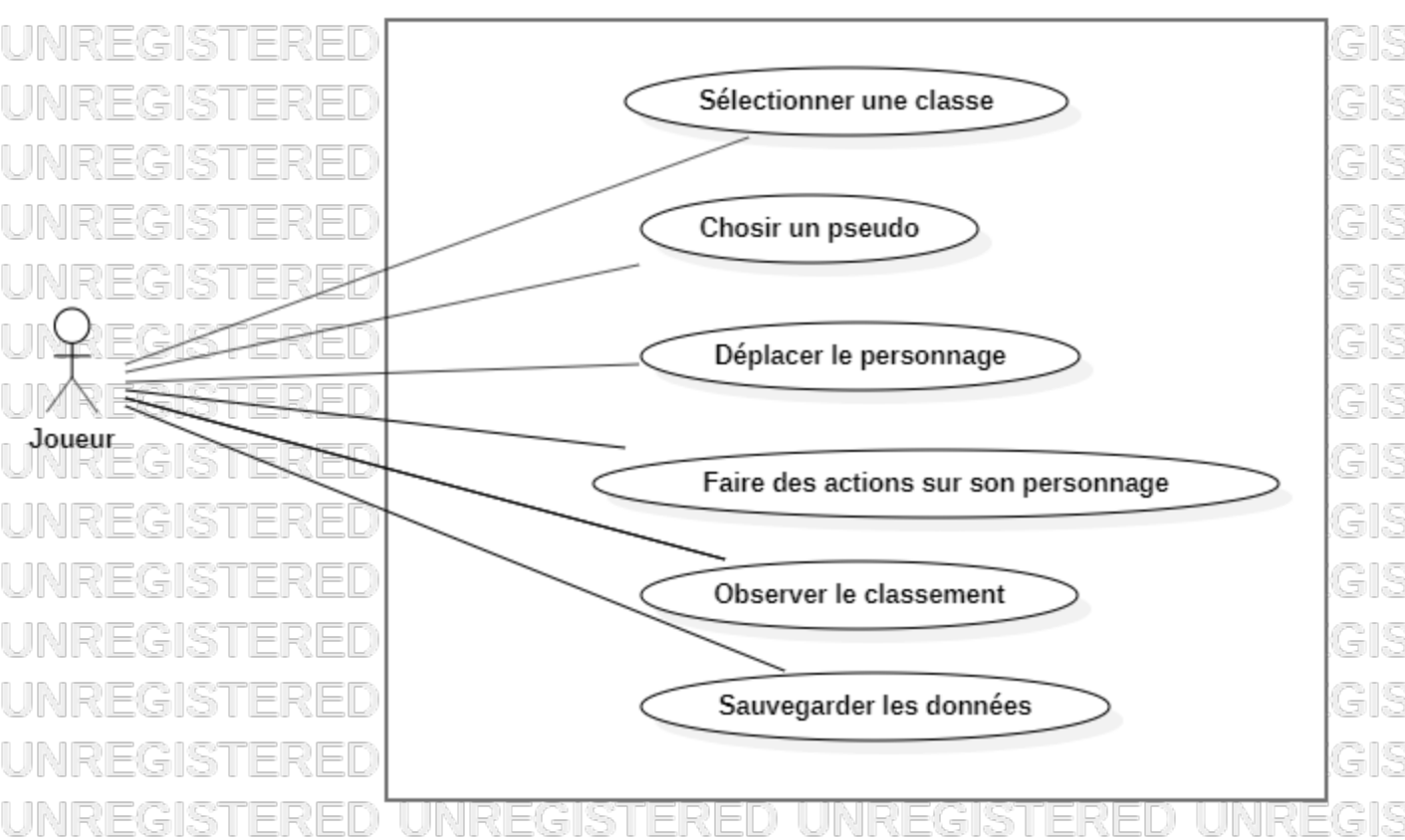
Fin de la manche :

- Si le joueur a gagné, une nouvelle vague apparaît et le joueur reprend sa place initiale avec le nombre de vie de la manche précédente, mais sans réinitialiser ces soins ni ses compétences et sans changer de pseudo ou de classe
- Si une vague d'ennemi gagne, le joueur revient à l'écran des scores (classement avec les anciens scores des joueurs), il peut rajouter son score désigné par son pseudo, la classe, le temps de jeu, et le nombre de vague battu

Classement :

- Affiche le tableau des scores avec les pseudos, la classe, les temps de jeu, et le nombre de vague battu





Compétences devant être maîtrisées

Je maîtrise la notion d'immuabilité de la classe String. [sur 0.5 point]

Un objet dit immuable est une instance de classe dont les membres exportés (ou visibles, que cela soit par un modificateur d'accès direct ou indirect, protected, public ou package private) ne peuvent être modifiés après création.

Un exemple classique d'objet immuable en Java est l'instanciation de la classe String :

```
String s = "ABC";  
s.toLowerCase();
```

La méthode toLowerCase() ne modifie pas la valeur "ABC" contenue dans la chaîne s. Au lieu de cela, un nouvel objet String est instancié, et la valeur "abc" lui est attribuée lors de sa construction. La méthode toLowerCase() retourne une référence à cet objet String. Si l'on souhaite que la chaîne s contienne la valeur "abc", il faut utiliser une approche différente :

```
s = s.toLowerCase();
```

Dans ce cas, la chaîne s référence un nouvel objet String qui contient "abc". Rien dans la déclaration d'un objet de classe String ne le contraint à être immuable : c'est plutôt qu'aucune des méthodes associées à la classe String n'affecte jamais la valeur d'un tel objet, ce qui le rend de fait immuable.

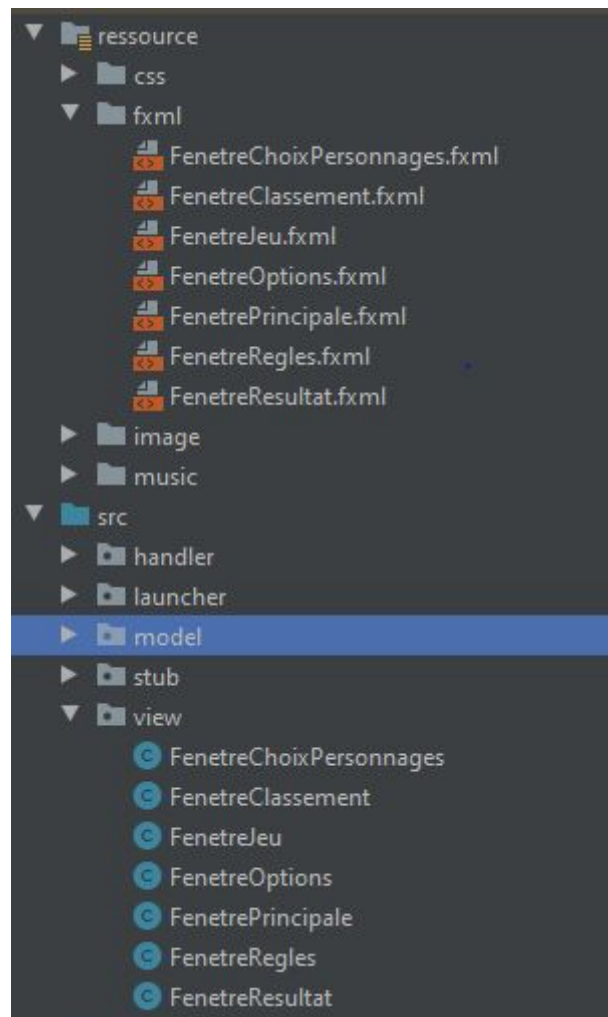
```
public Personnage deplacementPersonnage(Personnage joueur , String position){  
    int fin;  
    String columnS;  
    int positionX;  
    String rowS;  
    int positionY;  
    int ancienpositionX;  
    int ancienpositionY;  
    int difpositionX;  
    int difpositionY;  
    int ancienptsmouv;  
    int ptsmvt;  
    fin = position.indexOf('/');  
    columnS = position.substring(0, fin);  
    positionX = Integer.parseInt(columnS.trim());  
    rowS = position.substring(fin + 1);  
    positionY = Integer.parseInt(rowS.trim());
```

Je maîtrise les règles de nommage Java. [sur 1 point]

- Les noms des identificateurs ne peuvent pas utiliser les mots clés.
- Les noms des getters (accesseurs) doivent commencer par « get ».
- Les noms des setters (mutateurs) doivent commencer par « set ».

Entités	Règles	Exemple
Les packages	Toujours écrits tout en minuscules	com.entreprise.projet
Les classes, les interfaces et les constructeurs	<p>La première lettre est en majuscule.</p> <p>Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_'</p> <p>Le nom d'une classe peut finir par Impl pour la distinguer d'une interface qu'elle implémente.</p> <p>Les classes qui définissent des exceptions doivent finir par Exception.</p>	MaClasse MonInterface MaClasse()
Les méthodes	<p>Leur nom devrait contenir un verbe.</p> <p>La première lettre est obligatoirement une minuscule.</p> <p>Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule sans mettre de caractère underscore '_'</p> <p>Les méthodes pour obtenir la valeur d'un champ doivent commencer par get suivi du nom du champ.</p> <p>Les méthodes pour mettre à jour la valeur d'un champ doivent commencer par set suivi du nom du champ</p> <p>Les méthodes pour créer des objets (factory) devraient commencer par new ou create</p> <p>Les méthodes de conversion devraient commencer par to suivi par le nom de la classe renvoyée à la suite de la conversion</p>	public float calculerMontant()

Les variables	<p>La première lettre est obligatoirement une minuscule et ne devrait pas être un caractère dollar '\$' ou underscore '_' même si ceux-ci sont autorisés.</p> <p>Pour les variables d'instances non publiques, certains recommandent de commencer par un underscore pour éviter la confusion avec le nom d'une variable fournie en paramètre d'une méthode telle que le setter.</p> <p>Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_'.</p> <p>Les noms de variables composés d'un seul caractère doivent être évités sauf pour des variables provisoires (index d'une boucle).</p> <p>Les noms communs pour ces variables provisoires sont i, j, k, m et n pour les entiers et c, d et e pour les caractères.</p>	<pre>String nomPersonne; Date dateDeNaissance; int i;</pre>
Les constantes	<p>Toujours en majuscules, chaque mots est séparés par un underscore '_'. Ces variables doivent obligatoirement être initialisées lors de leur déclaration.</p>	<pre>static final int VAL_MIN = 0; static final int VAL_MAX = 9;</pre>



Convention de nommage des packages et des classes liées à la vue

Je sais binder bidirectionnellement deux propriétés JavaFX. [sur 1 point]

Parfois, il peut être utile d'établir des liaisons bidirectionnelles : c'est-à-dire des liaisons dans les deux sens. Si la propriété A et la propriété B sont liées entre elles par une liaison bidirectionnelle, alors tout changement de valeur de la propriété A se répercute automatiquement sur la propriété B ; mais tout changement de valeur de la propriété B se répercute automatiquement sur la propriété A également !



```
labelpseudo.textProperty().bindBidirectional(inputpseudo.textProperty());
```

Je sais binder unidirectionnellement deux propriétés JavaFX. [sur 1 point]

Si une propriété B est liée à une propriété A, alors tout changement de valeur de A est automatiquement répercuté sur la propriété B. De plus, tant que la liaison est active, il est impossible de modifier manuellement la valeur de la propriété B. Si B est en lecture seule, il est bien sûr totalement impossible de la lier sur A. Si A est en

lecture seule, elle peut sans problème être la source de liaisons effectuées par d'autres propriétés.

```
textProperty().bind(joueur.vagueProperty().asString());
```

Je sais coder une classe Java en respectant des contraintes de qualité de lecture de code. [sur 2 points]

```
public class Classement {

    private ObservableList<Joueur> joueursObs = FXCollections.observableArrayList();

    private ListProperty<Joueur> joueurs = new SimpleListProperty<>(joueursObs);

    public ListProperty<Joueur> joueursProperty() { return joueurs; }
    public ObservableList<Joueur> getJoueurs() { return joueurs.get(); }
    public void setJoueurs(ObservableList<Joueur> joueurs) { this.joueurs.set(joueurs); }

    /**
     * Add joueur.
     *
     * @param j the j
     */
    public void addJoueur(Joueur j) { joueursObs.add(j); }

}
```

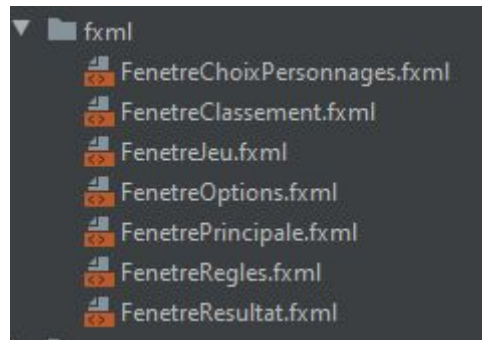
Je sais contraindre les éléments de ma vue, avec du binding FXML. [sur 2 points]

```
<ListView items="{listViewJoueur.items}" fx:id="listeViewJoueurTemps"/>
```

Je sais définir une CellFactory fabriquant des cellules qui se mettent à jour au changement du modèle. [sur 2 points]

```
listViewJoueur.setCellFactory(__ -> updateItem(joueur, empty) -> {
    super.updateItem(joueur, empty);
    if (!empty) {
        textProperty().bind(joueur.pseudoProperty());
    } else {
        textProperty().unbind();
        setText("");
    }
});
```

Je sais développer une application graphique en JavaFX en utilisant FXML. [sur 2 points]



```
<VBox xmlns:fx="http://javafx.com/fxml" fx:id="vboxback"
  style="-fx-background-image: url(image/background.jpg); -fx-background-repeat: no-repeat; alignment="CENTER" xmlns="http://javafx.com/javafx"
  fx:controller="view.FenetrePrincipale">
  <Button mnemonicParsing="false" text="Jouer" onAction="#afficherJouer">
    <VBox.margin>
      <Insets top="10.0"/>
    </VBox.margin>
  </Button>
  <Button mnemonicParsing="false" text="Règles du jeu" onAction="#afficherRegles">
    <VBox.margin>
      <Insets top="20.0"/>
    </VBox.margin>
  </Button>
  <Button mnemonicParsing="false" text="Classement" onAction="#afficherClassement">
    <VBox.margin>
      <Insets top="20.0"/>
    </VBox.margin>
  </Button>
  <Button mnemonicParsing="false" text="Options" onAction="#afficherOption">
    <VBox.margin>
      <Insets top="20.0"/>
    </VBox.margin>
  </Button>
  <Button mnemonicParsing="false" text="Quitter" onAction="#quitter">
    <VBox.margin>
      <Insets top="20.0"/>
    </VBox.margin>
  </Button>
</VBox>
```

Exemple du fichier de la fenêtre principale de notre application en FXML

Le FXML est un format de fichier propre à JavaFX et servant, entre autres, à définir des interfaces graphiques. Ce format utilise tout simplement une syntaxe XML

Je sais éviter la duplication de code. [sur 1 point]

La duplication de code arrive à la suite de la programmation par copier-coller. C'est une erreur classique de débutants en programmation informatique ?

Je sais hiérarchiser mes classes pour spécialiser leur comportement. [sur 2 points]

L'héritage possède deux caractéristiques pouvant parfois être en contradiction:

- l'extension de code, qui consiste à dériver une nouvelle classe d'une classe déjà existante, pour hériter du code de la classe existante et en ajouter sans avoir à tout réécrire.
- le polymorphisme qui modélise la relation dans laquelle un objet peut être utilisé à la place d'un autre objet.

```
public abstract class Entite
```

```

public class Ennemi extends Entite{

public abstract class Personnage extends Entite{

public class Guerrier extends Personnage {

public class Archer extends Personnage {

```

Je sais intercepter des événements en provenance de la fenêtre JavaFX. [sur 2 points]

Les événements utilisateurs sont gérés par plusieurs interfaces EventListener.

Les interfaces EventListener permettent de définir les traitements en réponse à des événements utilisateurs générés par un composant. Une classe doit contenir une interface auditrice pour chaque type d'événements à traiter :

- ActionListener : clic de souris ou enfoncement de la touche Enter
- ItemListener : utilisation d'une liste ou d'une case à cocher
- MouseMotionListener : événement de souris
- WindowListener : événement de fenêtre

```
buttonmvt.setOnAction(deplacementHandler);
```

```

public class DeplacementHandler extends GameHandler{

    private EventHandler<ActionEvent> deplacementHandler;
    private Personnage personnage;

    /**
     * Instantiates a new Deplacement handler.
     *
     * @param personnage the personnage
     */
    public DeplacementHandler(Personnage personnage) { this.personnage=personnage; }

    /**
     * Handle.
     */
    public void handle(){
        deplacementHandler = event -> {
            Personnage joueur = personnage;;
            String position = (((Button) event.getSource()).getText()).trim();
            joueur = joueur.deplacementPersonnage(joueur,position);
            if (joueur.getPts_mouvement() > -1) {
                setPartie(joueur);
            }
            personnage = joueur;
        };
    }
}

```

Je sais maintenir une encapsulation adéquate dans mes classes. [sur 2 points]

L'encapsulation désigne le principe de regroupement des données et du code qui les utilise au sein d'une même unité. On va très souvent utiliser le principe d'encapsulation afin de protéger certaines données des interférences extérieures en forçant l'utilisateur à utiliser les méthodes définies pour manipuler les données.

Classe Fenetreresultat :

Attributs :

```
@FXML
private TextField textpseudo;
@FXML
private TextField textnbvague;
@FXML
private TextField textnbtour;
@FXML
private TextField texttemps;
@FXML
private ListView<Joueur> listViewJoueur;
@FXML
private ListView<Joueur> listViewJoueurVague;
@FXML
private ListView<Joueur> listViewJoueurNbtour;
@FXML
private ListView<Joueur> listViewJoueurTemps;
@FXML
private Button boutonSave;

private Classement classementAfficher;
private String pseudo;
private String vague;
private String nbtour;
private String temps;
private Joueur joueur;
private String filename = "classement.json";
private String sourcePrincipale = "/FXML/FenetrePrincipale.fxml";
private int widthScene = 720;
private int heightScene = 480;
```

Méthodes :

```

public void initialize() throws IOException {
    chargerListViews();
}

private void chargerListViews() throws IOException {
    classementAfficher = LoadJSON.Load(filename);
    listViewJoueur.itemsProperty().bind(classementAfficher.joueursProperty());

    listViewJoueur.setCellFactory(__ -> updateItem(joueur, empty) -> {
        super.updateItem(joueur, empty);
        if (!empty) {
            textProperty().bind(joueur.pseudoProperty());
        } else {
            textProperty().unbind();
            setText("");
        }
    });
}

```

```

public void accueil(ActionEvent actionEvent) throws IOException {

```

```

public void saveJoueur(ActionEvent actionEvent) throws IOException {












```

Je sais maintenant, dans un projet, une responsabilité unique pour chacune de mes classes. [sur 2 points]

Le principe de responsabilité unique comme suit : « une classe ne doit changer que pour une seule raison »

Prenons l'exemple d'un module qui compile et imprime un rapport. Imaginons que ce module peut changer pour deux raisons. D'abord, le contenu du rapport peut changer. Ensuite, le format du rapport peut changer. Ces deux choses changent pour des causes différentes; l'une substantielle, et l'autre cosmétique. Le principe de responsabilité unique dit que ces deux aspects du problème ont deux responsabilités distinctes, et devraient donc être dans des classes ou des modules séparés. Ce serait une mauvaise conception de coupler ces deux choses dans une même classe.

La raison pour laquelle il est important de garder une classe axée sur une seule préoccupation est que cela rend la classe plus robuste. En continuant avec l'exemple précédent, s'il y a un changement dans le processus de compilation du rapport, il y a un plus grand danger que le code d'impression se casse si elle fait partie de la même classe.

-  Archer
-  Classement
-  Ennemi
-  Entite
-  Guerrier
-  Instru
-  ISave
-  Joueur
-  LoadJSON
-  Personnage
-  SaveJSON

Ici toutes les classes du modèle, ayant chacune une responsabilité unique

Je sais gérer la persistance de mon modèle. [sur 2 points]

La persistance des données et parfois des états d'un programme réfère au mécanisme responsable de la sauvegarde et de la restauration des données. Ces mécanismes font en sorte qu'un programme puisse se terminer sans que ses données et son état d'exécution ne soient perdus.

```
public void save(String nomfic, Classement classement) {
    JSONArray joueurList = new JSONArray();
    for (Joueur j : classement.getJoueurs()) {
        JSONObject joueurDetails = new JSONObject();
        joueurDetails.put("pseudo", j.getPseudo());
        joueurDetails.put("vague", j.getVague());
        joueurDetails.put("nbtour", j.getNbtour());
        joueurDetails.put("tempsJ", j.getTempsJ());
        JSONObject joueurObject = new JSONObject();
        joueurObject.put("Joueur", joueurDetails);
        joueurList.add(joueurObject);
    }
    try (FileWriter file = new FileWriter(nomfic)) {
        file.write(joueurList.toJSONString());
        file.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Persistance faite en JSON

Je sais surveiller l'élément sélectionné dans un composant affichant un ensemble de données. [sur 2 points]

```
sonSlider.valueProperty().addListener((observable, oldValue, newValue) -> volume = (newValue.floatValue()) / 100);  
sonSlider.valueProperty().addListener((observable, oldValue, newValue) -> setmusicMP3(mon_mediaPlayer, volume: (newValue.floatValue()) / 100));
```

Je sais utiliser à mon avantage le polymorphisme. [sur 2 points]

Le polymorphisme veut dire que le même service, aussi appelé opération ou méthode, peut avoir un comportement différent selon les situations.

```
public Personnage reinitiliaserPersonnage(Personnage joueur) {  
    Guerrier guerrier = new Guerrier( Ppseudo: "", PpositionX: 0, PpositionY: 0);  
    Archer archer = new Archer( Ppseudo: "", PpositionX: 0, PpositionY: 0);  
    Class<? extends Personnage> pclass = joueur.getClass();  
    if (pclass == guerrier.getClass()) {  
        joueur.setPts_mouvement(guerrier.getPts_mouvement());  
        joueur.setDistance(guerrier.getDistance());  
        joueur.setPts_action(guerrier.getPts_action());  
    }  
    if (pclass == archer.getClass()) {  
        joueur.setPts_mouvement(archer.getPts_mouvement());  
        joueur.setDistance(archer.getDistance());  
        joueur.setPts_action(archer.getPts_action());  
    }  
    return joueur;  
}
```

Je sais utiliser certains composants simples que me propose JavaFX. [sur 0.5 point]

```
public void setPartie(Personnage personnage){  
    Node node = gridplateau.getChildren().get(0);  
    gridplateau.getChildren().clear();  
    gridplateau.getChildren().add( index: 0, node);  
    this.personnage = personnage;  
    labelpseudo.setText(this.personnage.getPseudo() + " (LVL:" + this.personnage.getLvl() + ")");  
    labelvague.setText(String.valueOf(this.vague));  
    labelnbtour.setText("Nombre de tour jouer : " + this.nbtour);  
    labelptmouv.setText("Points de mouvement : " + this.personnage.getPts_mouvement());  
    labelptact.setText("Points d'action : " + this.personnage.getPts_action());  
    labelvie.setText(this.personnage.getPts_vie() + " / " + this.personnage.getVietotal() + " PV");  
    buttonsoin.setText("Soin (" + this.personnage.getNb_potion() + ")");  
    buttoncompetence.setText(this.personnage.getCompetence());  
    imagejoueur(this.personnage);  
    imageennemie(this.ennemi);  
    vague(this.vague, this.personnage, this.ennemi);  
}
```

Je sais utiliser certains layout que me propose JavaFX. [sur 1 point]

Si vous connaissez le langage Java classique, vous savez sûrement déjà ce qu'est un layout. C'est une structure graphique dans laquelle vous pouvez ranger toutes sortes d'objets graphiques de façon bien organisée. En JavaFX, il existe huit types de layouts :

- 1. Le **BorderPane** qui vous permet de diviser une zone graphique en cinq parties : top, down, right, left et center.
- 2. La **Hbox** qui vous permet d'aligner horizontalement vos éléments graphiques.
- 3. La **VBox** qui vous permet d'aligner verticalement vos éléments graphiques.
- 4. Le **StackPane** qui vous permet de ranger vos éléments de façon à ce que chaque nouvel élément inséré apparaisse au-dessus de tous les autres.
- 5. Le **GridPane** permet de créer une grille d'éléments organisés en lignes et en colonnes
- 6. Le **FlowPane** permet de ranger des éléments de façon à ce qu'ils se positionnent automatiquement en fonction de leur taille et de celle du layout.
- 7. Le **TilePane** est similaire au FlowPane, chacune de ses cellules fait la même taille.
- 8. L'**AnchorPane** permet de fixer un élément graphique par rapport à un des bords de la fenêtre : top, bottom, right et left.

```
@FXML
private GridPane gridplateau;
```

```
Node node = gridplateau.getChildren().get(0);
gridplateau.getChildren().clear();
gridplateau.getChildren().add(0, node);
```

Je sais utiliser GIT pour travailler avec mon binôme sur le projet. [sur 2 points]

Git est un système de contrôle de version open source gratuit, créé par Linus Torvalds, en 2005, chaque développeur dispose en local de l'historique complet du dépôt de son code. Ceci ralentit le clone initial du dépôt, mais accélère considérablement les opérations ultérieures (commit, diff, merge et log).

root @ master URL : <https://forge.clermont-universite.fr/git/projet06-javafx-2019> | Statistiques | Branche: master | Révision:

	Nom	Taille
Documentation		
Projet		
README.md		249 octet

Dernières révisions

#	Date	Auteur	Commentaire
7fbc353d	19/01/2020 19:56	Corentin Cousse	Merge entre les branches, 3ème version stable de l'application, mise à jour de la qualité du code
a9e9d7c2	19/01/2020 19:33	Corentin Cousse	Merge entre les branches, 2ème version stable de l'application, mise à jour de la qualité du code
e1671c25	18/01/2020 19:46	Corentin Cousse	Merge entre les branches, 1ère version stable de l'application
e96b3453	13/01/2020 11:43	Corentin Cousse	Mise à jour du Diag de Classes et ajout d'un readme
d9ecf6d	27/11/2019 15:21	Alexandre BOUGRAT	Modification DiagClasses
4e9d1e1b	22/11/2019 15:44	Alexandre BOUGRAT	nouvelle branche
b356069c	20/11/2019 15:18	Myogamevideo	Ajout diagramme de classe
4187aa48	20/11/2019 15:15	Alexandre BOUGRAT	Renommage package
86b1a174	20/11/2019 14:36	Myogamevideo	Ajout cas utilisation
5099b643	20/11/2019 14:15	Myogamevideo	Ajout context

Je sais utiliser le type statique adéquat pour mes attributs ou variables. [sur 1 point]

On appelle élément statique d'une classe tout élément attaché à cette classe plutôt qu'à l'une de ses instances. Un élément statique peut exister, être référencé, ou s'exécuter même si aucune instance de cette classe n'existe. Un élément statique ne peut référencer this.

Il est possible de définir quatre types d'éléments statiques :

- des champs statiques : la valeur de ce champ est la même pour tous les objets instance de la classe ;
- des méthodes statiques ;
- des blocs statiques ;
- des classes membre statiques.

```
private static void parseClassementObject(JSONObject joueurObject, Classement classement)
```

Je sais utiliser les collections. [sur 1 point]

Les collections sont des objets qui permettent de gérer des ensembles d'objets. Ces ensembles de données peuvent être définis avec plusieurs caractéristiques : la possibilité de gérer des doublons, de gérer un ordre de tri, etc. ...

Une collection est un regroupement d'objets qui sont désignés sous le nom d'éléments.

L'API Collections propose un ensemble d'interfaces et de classes dont le but est de stocker de multiples objets. Elle propose quatre grandes familles de collections, chacune définie par une interface de base :

- List : collection d'éléments ordonnés qui accepte les doublons
- Set : collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons
- Map : collection sous la forme d'une association de paires clé/valeur
- Queue et Deque : collections qui stockent des éléments dans un certain ordre avant qu'ils ne soient extraits pour traitement

```

public class Classement {

    private ObservableList<Joueur> joueursObs = FXCollections.observableArrayList();

    private ListProperty<Joueur> joueurs = new SimpleListProperty<>(joueursObs);

    /**
     * Joueurs property list property.
     *
     * @return the list property
     */
    public ListProperty<Joueur> joueursProperty() { return joueurs; }

    /**
     * Gets joueurs.
     *
     * @return the joueurs
     */
    public ObservableList<Joueur> getJoueurs() { return joueurs.get(); }

    /**
     * Sets joueurs.
     *
     * @param joueurs the joueurs
     */
    public void setJoueurs(ObservableList<Joueur> joueurs) { this.joueurs.set(joueurs); }

    /**
     * Add joueur.
     *
     * @param j the j
     */
    public void addJoueur(Joueur j) { joueursObs.add(j); }
}

```

Je sais utiliser les différents composants complexes (listes, combo...) que me propose JavaFX. [sur 1 point]

```

comboBox.getItems().add("Dark");
comboBox.getItems().add("Light");
comboBox.getSelectionModel().selectFirst();

```

Je sais utiliser les lambda-expression. [sur 1 point]

Java est un langage orienté objet : à l'exception des instructions et des données primitives, tout le reste est objets, même les tableaux et les chaînes de caractères. Java ne propose pas la possibilité de définir une fonction/méthode en dehors d'une classe ni de passer une telle fonction en paramètre d'une méthode. Depuis Java 1.1, la solution pour passer des traitements en paramètres d'une méthode est d'utiliser les classes anonymes internes.

Pour faciliter, entre autres, cette mise à oeuvre, Java 8 propose les expressions lambda. Les expressions lambda sont aussi nommées closures ou fonctions anonymes : leur but principal est de permettre de passer en paramètre un ensemble de traitements.

De plus, la programmation fonctionnelle est devenue prédominante dans les langages récents. Dans ce mode de programmation, le résultat de traitements est décrit mais pas la façon dont ils sont réalisés. Ceci permet de réduire la quantité de code à écrire pour obtenir le même résultat.

```
listeViewJoueurVague.setCellFactory(__ -> updateItem(joueur, empty) -> {
```

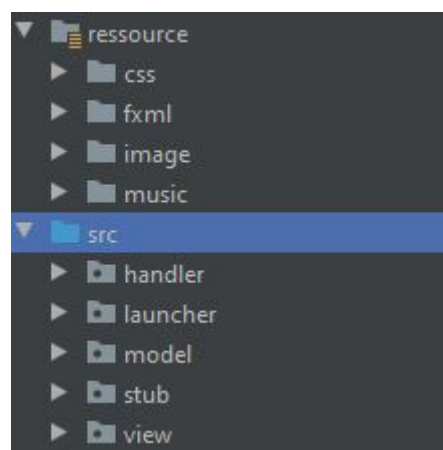
Je sais utiliser les listes observables de JavaFX. [sur 1 point]

Les listes observables sont partout présentes dans JavaFX, elles sont littéralement à la base du concept du SceneGraph qui permet d'afficher des nœuds graphiques dans une interface utilisateur. En effet chaque nœud parent, groupe ou gestionnaire de mise en page dispose d'une liste observable de nœuds enfants. Au niveau des contrôles, ces listes sont également utilisées par les boîtes déroulantes et autres listes, arbres et tables graphiques.

De base, ces listes observables permettent de recevoir, via des écouteurs de type `ChangeListener`, des événements de notification classiques tels que d'ajout, retrait, remplacement ou encore permutation des éléments contenus dans une de ces listes. Bref, tout ce qui se rapporte aux manipulations classiques d'une liste. Cependant, il existe un autre type de notification : la notification de mise à jour qui permet d'être averti lorsque la propriété observable d'un élément de la liste est modifiée. Cette fonctionnalité peu connue permet d'être averti que l'état d'un des objets de la liste a changé ! Elle nécessite l'utilisation d'un nouveau concept : l'extracteur de propriétés.

```
private ObservableList<Joueur> joueursObs = FXCollections.observableArrayList();
```

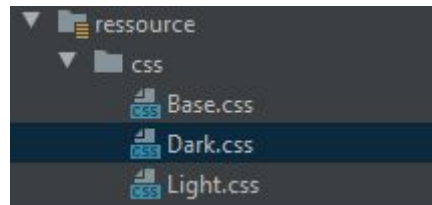
Je sais utiliser les packages à bon escient dans un projet. [sur 1 point]



Je sais utiliser un convertisseur lors d'un bind entre deux propriétés JavaFX. [sur 1 point]

Je sais utiliser un fichier CSS pour styler mon application JavaFX. [sur 1 point]

Cascading Style Sheets (CSS) est un langage de feuille de style utilisé pour décrire la présentation d'un document écrit en HTML ou en XML. CSS décrit la façon dont les éléments doivent être affichés à l'écran, sur du papier, en vocalisation, ou sur d'autres supports.



Je sais utiliser un formateur lors d'un bind entre deux propriétés JavaFX. [sur 1 point]

```
textProperty().bind(joueur.nbtourProperty().asString());
```