Liste de compétences

Par BOUDIAUX Pierre et DEVIENNE Thomas.

Je maîtrise les règles de nommage Java.

Le nom des variables est écris en camel case (tout en minuscule sauf la première lettre des mots en interne du nom de variable, comme par exemple nomDeVariable) ou avec des underscores entre chaque mots (comme nom_de_variable). La première lettre des noms de classes et des interface est en majuscule. Les constantes statiques sont en majucules intégralement.

```
38  /**
39   * Classe qui gere le deplacement du personnage (le Dino), herite de Deplaceur
40   */
41  public class DeplaceurBasePerso extends Deplaceur {
42
43      /**
44       * Gravite applique sur le personnage
45       */
46      private final double GRAVITE = 9.81;
47
48      /**
49       * Velocite du personnage
50       */
51      private static int velocity = 5;
52
```

Je sais binder bidirectionnellement deux propriétés JavaFX.

Dans view.FenetreMenu.java :

```
142     nomJoueur.set(Variables.nomJoueur);
143     nomJoueurTF.textProperty().bindBidirectional(nomJoueur);
```

Je sais binder unidirectionnellement deux propriétés JavaFX.

```
78          // binding des proprietes
79          dino_view.xProperty().bind(manager.getMonde().getDino().posXProperty());
80          dino_view.yProperty().bind(manager.getMonde().getDino().posYProperty());
81          nb_de_pets.textProperty().bind(manager.getMonde().getDino().nbPetsStringProperty());
```

Je sais coder une classe Java en respectant des contraintes de qualité de lecture de code.

```java
/**
 * Item qui recharge la vie du Dinosaure, Herite de Item et implemente IBonus
 */
public class ItemRechargeVie extends Item implements IBonus {


    /**
     * Constructeur de ItemAugmenterSaut
     * @param x Position en X
     * @param y Position en Y
     * @param hitbox Hitbox de l'item
     */
    public ItemRechargeVie(double x, double y, Rectangle hitbox) {
        super(x, y, hitbox);
        setImageView(new ImageView(new Image("file:///" + System.getProperty("user.dir") + "/rsrc/media/coeur.png")));
    }


    /**
     * Ajoute le Bonus au Dinosaure
     * @param d Dinosaure affecte
     * @param value Nombre de vie a ajouter au Dinosaure
     */
    @Override
    public void addBonus(Dino d, int value) {
        if(d.getPdv()<10) d.setPdv(d.getPdv() + value);
    }

    @Override
    public String toString() {
        return "ItemRechargeVie{" +
                "posX=" + posX +
                ", posY=" + posY +
                '}';
    }
}
```

Nous mettons toujours en premier les attributs, ensuite le constructeur, puis les méthodes de la classe. En dernier nous redéfinissons les méthodes héritées si besoin.

Je sais contraindre les éléments de ma vue, avec du binding FXML.

Dans view.FenetreJeu.java :

```java
        // binding des proprietes
        dino_view.xProperty().bind(manager.getMonde().getDino().posXProperty());
        dino_view.yProperty().bind(manager.getMonde().getDino().posYProperty());
        nb_de_pets.textProperty().bind(manager.getMonde().getDino().nbPetsStringProperty());
```

Ici nous contraignons la position X et Y de notre ImageView dino_view dans le code behind de notre FenetreJeu.

Je sais définir une CellFactory fabriquant des cellules qui se mettent à jour au changement du modèle.

Dans view.FenetreMenu.java :

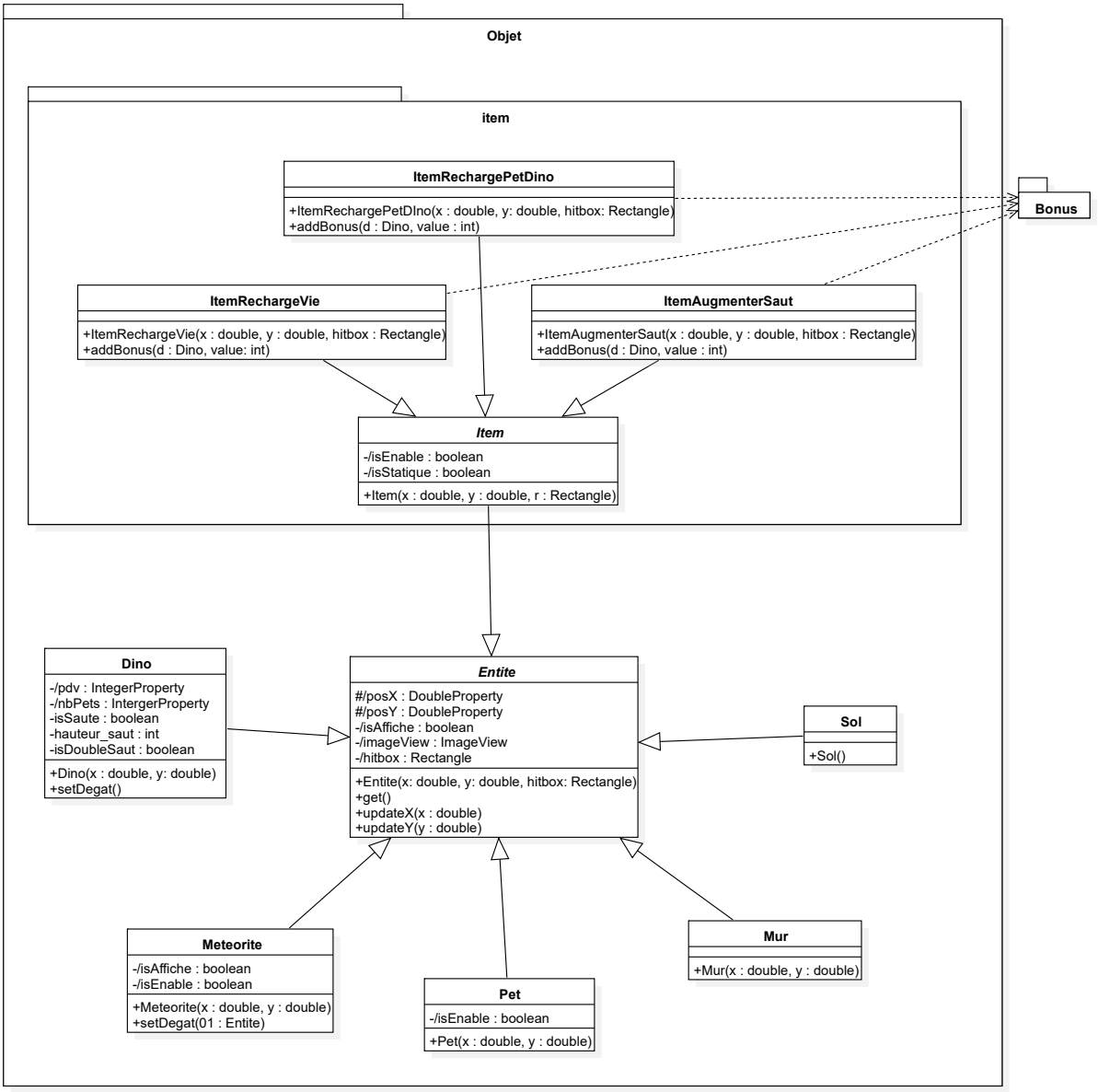
```java
        TableColumn joueurCol = new TableColumn("Joueur");
        joueurCol.setMinWidth(290);
        TableColumn scoreCol = new TableColumn("score");
        scoreCol.setMinWidth(290);
        scoreTableView.getColumns().add(joueurCol);
        scoreTableView.getColumns().add(scoreCol);

        scoreCol.setCellValueFactory(new PropertyValueFactory<ResultatScore, Long>("score"));
        joueurCol.setCellValueFactory(new PropertyValueFactory<ResultatScore, String>("nomJoueur"));

        scoreTableView.setItems(allscores);
        scoreCol.setSortType(TableColumn.SortType.DESCENDING);
        scoreTableView.getSortOrder().add(scoreCol);
```

Je sais éviter la duplication de code.

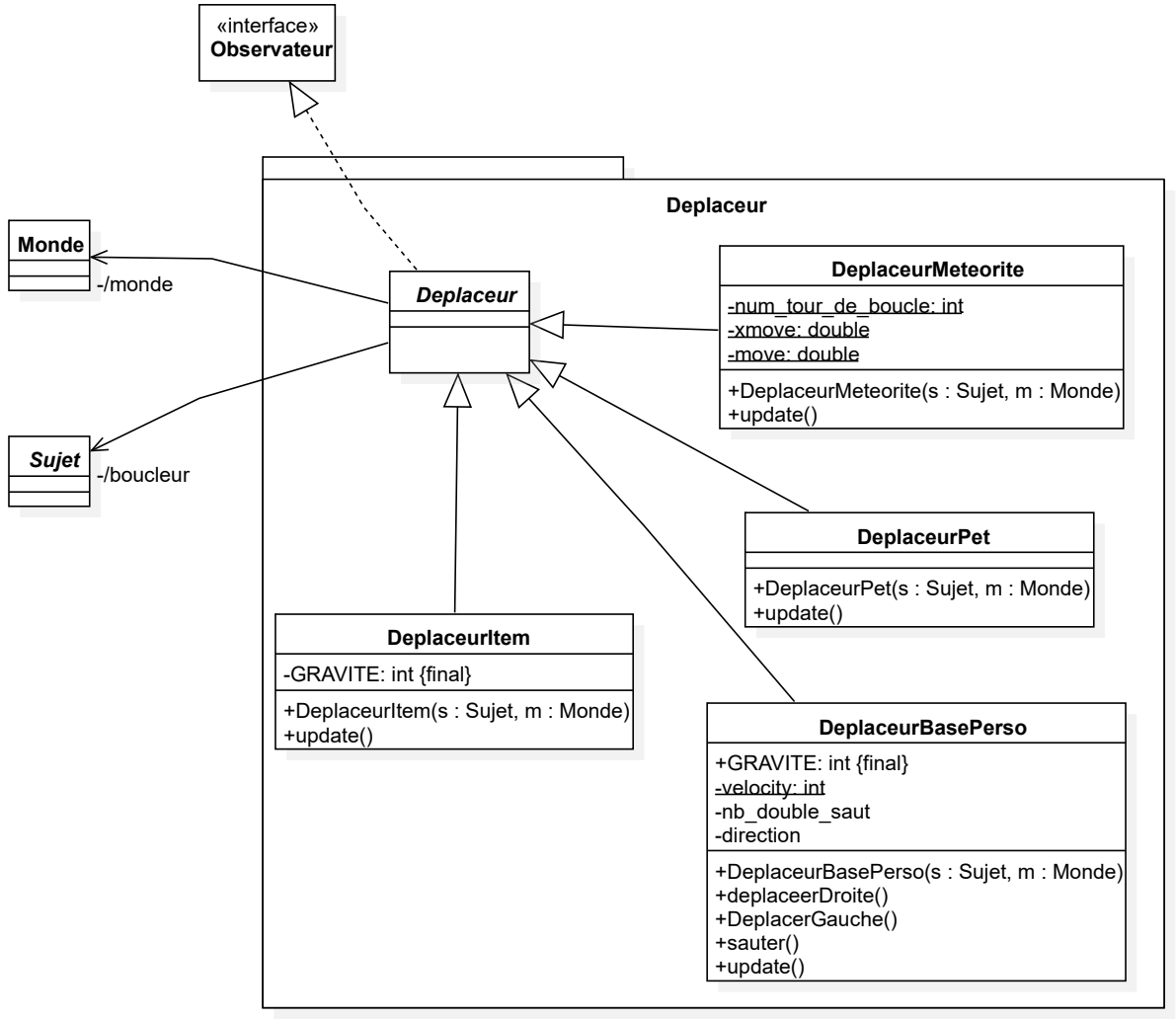
Ci-dessous le diagramme de classe du package objet :



En réalisant des relations d'héritages entre classes, on évite de dupliquer les attributs et les méthodes dans les classes filles en plus de rendre notre code plus flexible (passage de paramètres plus robustes en cas de modification de types instanciés, etc.).

Je sais hiérarchiser mes classes pour spécialiser leur comportement.

Ci-dessous le diagramme de classe du package Deplaceur :



Nous avons une classe abstraite Deplaceur qui sert de modèle pour les autres Déplaceurs. Cette classe abstraite défini toutes les méthodes et les attributs communs aux déplaceurs. Les autres déplaceurs qui hérite de cette classe récupère donc les caractéristiques d'un déplaceur et ajoutent des attributs ou méthodes en rapport avec leur spécialisation. Ils peuvent aussi redéfinir les méthodes de la classe mère.

Je sais intercepter des évènements en provenance de la fenêtre JavaFX.

Dans view.FenetreJeu.java :

```
93      @FXML
94      public void handleKeyPressed(KeyEvent keyEvent) {
95          controledep.setText("");
96          switch (keyEvent.getCode()) {
97              case D, RIGHT -> {
98                  manager.deplacerDinoDroite();
99                  dino_view.setScaleX(1f);
100             }
101             case Q, LEFT -> {
102                 manager.deplacerDinoGauche();
103                 dino_view.setScaleX(-1f);
104             }
105             case S, DOWN -> {
106                 manager.creerPet();
107             }
108             case SPACE -> {
109                 manager.sauter();
110             }
111         }
112     }
```

Je sais maintenir, dans un projet, une responsabilité unique pour chacune de mes classes.

Chaque classe possède une responsabilité propre.

« Chaque package possède, comme chaque classe, une responsabilité qui lui est propre. Ainsi on a :

– Manager : fournir une interface unique pour manipuler et gérer des instances d'objets du modèle ;
– Objet : regroupe toutes les classes qui représente des objets ;
– Bonus : regroupe des interfaces qui fournissent un modèle commun aux bonus ;
– Loader : regroupe le loader (le point de démarrage du programme) ;
– Persistance : gère la persistance du programme, c'est-à-dire la sauvegarde et le chargement des données ;
– Score : regroupe les classes qui concernent les scores utilisateur ;
– Déplaceur : regroupe toutes les classes dont le rôle est de déplacer des objets ;
– Collisionneur : regroupe toutes les classes dont le rôle est de vérifier la collision entre deux objets ;
– Boucleur : classes chargées de boucler de manière periodique toutes les x ms ;
– Créateur : regroupe toutes les classes dont le rôle est de créer des objets ; », Compte-rendu

Je sais gérer la persistance de mon modèle.

La persistance est gérée dans le package Persistance dans le Modele. Voici la classe qui permet de sauver des scores :

```java
@Override
public void SaveScore(SortedList<ResultatScore> allscore) throws Exception {

    try (FileOutputStream fos = new FileOutputStream("bin/scores.bin");
         ObjectOutputStream oos = new ObjectOutputStream(fos))
    {
        /*
        if(allscore.size()<=5){/// si il y a moin de 5 scores
            for (ResultatScore res : allscore
            ) {
                System.out.println(res);
                oos.writeObject(res);
            }
        }

        else {
            for(int i =1; i<=5;i++){ // on sauvegarde que les 5 meilleurs scores
                System.out.println(allscore.get(allscore.size()-i));
                oos.writeObject(allscore.get(allscore.size()-i));
            }
        }*/
        for (ResultatScore res : allscore
        ) {
            oos.writeObject(res);
        }

    }

    catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

Et celle qui permet de charges des scores :

```java
/**
 * methode de chargement des donnees
 * @return liste des scores
 */
@Override
public List<ResultatScore> LoadScore() {
    List<ResultatScore> allRes = new ArrayList<>();
    int nblu = 0;
    try {
        FileInputStream fis = new FileInputStream("bin/scores.bin");
        ObjectInputStream ois = new ObjectInputStream(fis);

            /*
            while (nblu < 5) { // on charges que les 5 premier scores (les meilleurs)
                try {
                    allRes.add((ResultatScore) ois.readObject());
                    nblu++;
                } catch (ClassNotFoundException e) {
                    return allRes;
                } catch (EOFException e) {
                    return allRes;
                }
            }*/
        while (true){
            allRes.add((ResultatScore) ois.readObject());
        }


    }
    catch (FileNotFoundException e) {
        System.out.println("Fichier de score inexistant");
    }
    catch (EOFException e){
        return allRes;
    }
    catch (IOException e) {
```

Je sais utiliser à mon avantage le polymorphisme.

Dans la classe Monde.java :

```java
public void removeEntite(Entite e) {
    allEntite.remove(e);
    e.setPosX(-8000);
}
```

On manipule ici des Items / Météorites / Pets / etc. qui héritent de Entite.

Je sais utiliser GIT pour travailler avec mon binôme sur le projet.

Se référer à https://gitlab.iut-clermont.uca.fr/pibourdiau/meteorsurvive/-/commits/main pour voir les commits de chacun.

Je sais utiliser le type statique adéquat pour mes attributs ou variables.

Dans le modele.createur.CreateurPet.java :

```java
    /**
     * Constructeur d'un CreateurPet
     * @param m Monde où le createur vas creer
     * @param b Boucleur qui notifiera pour l'update
     */
    public CreateurPet(Monde m, Boucleur b) {
        super(m);
        boucleur = b;
        boucleur.attacher(this);
        nb_meteorite = Variables.NB_METEORITES_POUR_UN_PET;
    }
```

Nous avons utilisé le type statique le plus « bas » pour assurer le passage en paramètre d'un boucleur au constructeur de la classe CreateurPet. Ainsi, tout les types instanciés qui héritent de Boucleur sont acceptés par ce constructeur car on est assuré que ces types possèdent au moins toutes les méthodes de Boucleur.

De manière plus générale, on met à gauche (type statique) le type le plus « bas » et en type instancié le type voulu, ce dernier devant obligatoirement hérité du type statique ou l'implémenter.

Je sais utiliser les différents composants complexes (listes, combo…) que me propose JavaFX.

Dans view.FenetreMenu.java, nous utilisons un composant Vbox :

```java
    @FXML
    public VBox vBox;
```

Je sais utiliser les lambda-expression.

Dans view.FenetreJeu.java :

```java
        manager.getItems().addListener((InvalidationListener) observable -> {
            for (var item : manager.getItems()) {
                creerUIElement(item);
            }
        });
```

Je sais utiliser les listes observables de JavaFX.

Dans objet.Monde.java

```java
    /**
     * Liste de toutes les meteorites
     */
    private final ObservableList<Meteorite> listMeteorite = FXCollections.observableArrayList();

    /**
     * Listes de tous les pets
     */
    private final ObservableList<Pet> listPets = FXCollections.observableArrayList();

    /**
     * Liste des tous les Items
     */
    private final ObservableList<Item> listItem = FXCollections.observableArrayList();
```

On ajoute des éléments avec set(). On accède avec get(). On peux ajouter un listener sur ces éléments avec addListener comme ici dans view.FenetreJeu.java :

```java
        manager.getItems().addListener((InvalidationListener) observable -> {
            for (var item : manager.getItems()) {
                creerUIElement(item);
            }
        });
```

Je sais utiliser un convertisseur lors d'un bind entre deux propriétés JavaFX.

Dans view.FenetreJeu.java :

```java
    public void getScene() {
        pointVie.textProperty().bind(Bindings.convert(manager.getMonde().getDino().pdvProperty()));
        pointVie.setFont(Font.font("Impact", 20));
```

Je sais utiliser un fichier CSS pour styler mon application JavaFX.

Dans les ressources, le fichier de style StyleFenetre.css :

```css
.hboxStyle {
    -fx-padding: 10;
}

.playButton{
    -fx-font: 22 Impact;
     -fx-base: #b6e7c9;
}

.tableauScore{
    /*-fx-background-color: LightGrey;*/
}

.TitreColScore{
    -fx-font: 22 Impact;
     -fx-base: Black;
     -fx-stroke : White;
}
```

On fais appel à ce fichier comme ceci dans view.FenetreJeu.fxml :

```xml
<AnchorPane fx:id="fenetrejeu"
            xmlns="http://javafx.com/javafx/11.0.2"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="view.FenetreJeu"
            stylesheets="@style/StyleFenetreJeu.css">
```

Je sais intégrer, à bon escient, dans mon jeu, une boucle temporelle observable.

Dans modele.Boucleur.BoucleurJeu.java :

```java
public class BoucleurJeu extends Boucleur {


    public BoucleurJeu() {System.out.println("JEU");}

    /**
     * faire un boucle de 10ms, il y a donc 100 rafraichissement par secondes
     */
    @Override
    public void run() {
        Runnable notifieur = new Runnable() {
            @Override
            public void run() {
                notifier();
            }
        };
        // si le joueur veut bouger, on boucle
        while (true) {
            if(!isGameOver()){
                try {
                    sleep(10);
                    Platform.runLater(notifieur);
                } catch (InterruptedException e) {
                    //e.printStackTrace();
                }
            }


        }
    }
}
```

Ce boucleur est en charge de notifier les objets qui sont abonnés à ces notifications toutes les 10 ms. Ceci permet par exemple au DeplaceurBasePerso de savoir quand déplacer le joueur. En effet, sans cela la vitesse de déplacement du joueur serait dépendante de la vitesse du CPU de la machine du joueur, tandis qu'ici elle est dépendante de la vitesse du boucleur qui est la même pour chaque joueur. On abonne le déplaceur comme ceci dans le manager.Manager.java :

```java
    public Manager() {

        Boucleur.setGameOver(false);// debut d'une nouvelle partie
        monde = new Monde(this);
        boucleur = new BoucleurJeu();
        boucleM = new BoucleurMeteorite();

        deplaceurMeteorite = new DeplaceurMeteorite(boucleur, monde);
        deplaceurPet = new DeplaceurPet(boucleur, monde);
        deplaceurBasePerso = new DeplaceurBasePerso(boucleur, monde);
        deplaceurItem = new DeplaceurItem(boucleur, monde);

        threadJeu =  new Thread(boucleur);
        threadJeu.start();

        threadBoucleM = new Thread(boucleM);
        threadBoucleM.start();

        new CreateurMeteorite(monde, boucleM);
        new CreateurItem(monde, boucleM);
        new CreateurPet(monde, boucleM);

        score = new GestionnaireScore(boucleur);

    }
```

On observe aussi qu'on lance les boucleurs dans des Threads qui fonctionnent en parallèle de l'exécution du programme. Le second boucleur est celui des météorites qui plus lent que le boucleur de jeu.

Pour rendre nos boucleurs capables de notifier les éléments qui y sont abonnés et ces dernier à recevoir les notifications, nous avons implémenter dans notre projet le patron de conception Observateur/Observable.