

IF13YO10 Projet de programmation

Rapport Final

Slice & Dice

Réalisé par le groupe RClb :

Gabriel FISCHER
Michaël JOHNSON
Mila POPOVIĆ
Andrea TOMAS
Kai WITZMANN

SOMMAIRE

Introduction.....	Page 3
Organisation.....	Page 4
Cahier des charges, diagramme/organisation des classes.....	Page 5
Modèle de jeu, Entités, Héros, Ennemis.....	Page 5
Lancement de la partie.....	Page 8
Déroulement d'une partie, Choix des capacités.....	Page 9
Seulement des capacités pour gagner ?.....	Page 11
Équipements.....	Page 11
Extension-Boutique d'équipements.....	Page 12
Sorts.....	Page 14
Extension-Poison.....	Page 18
Extension-Évènements aléatoires.....	Page 19
Améliorations.....	Page 20
Différents menus.....	Page 23
Extensions - Boutique de capacités, Sauvegarde, Succès....	Page 26
Extension - Custom.....	Page 28
Développement du projet.....	Page 33
Dernier mot.....	Page 35
Annexes.....	Page 36

Introduction :

Slice & Dice est un jeu au tour par tour dans lequel le joueur contrôle une formation de plusieurs héros, chacun possédant un dé contenant des capacités qui seront tirées au hasard à chaque tour. L'objectif est d'avancer jusqu'au dernier niveau, en battant tous les ennemis sur le chemin.

Plus de détails :



Capture d'écran du jeu : Le joueur a le droit à 3 lancers de dés afin de choisir les capacités qui lui conviennent le mieux

Ce compte rendu suivra nos étapes de conception, de programmation de jeu, et les détails ainsi que les différentes fonctionnalités seront expliquées dans l'ordre dans lequel nous les avons travaillés. Cela permettra de garder une chronologie cohérente.

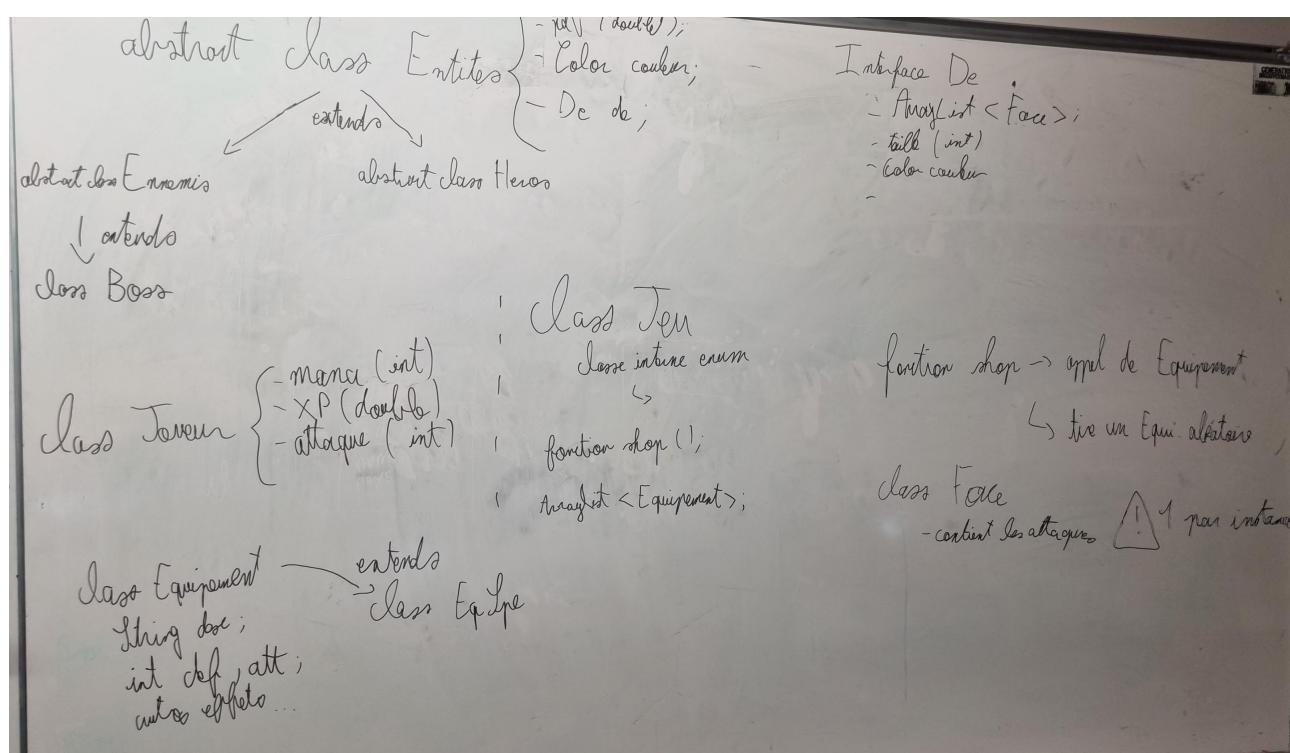
Ainsi, la première étape de la conception d'un jeu, c'est l'établissement d'un workflow.

Organisation :

Les détails sont sur chacun des comptes rendus hebdomadaires présents dans le dossier Documentation sur le git.

Dans les grandes lignes, nous avons choisi d'avoir des rendez-vous entre membres du groupe, hebdomadaires, juste après le rendez-vous avec notre professeur référent. Ces rendez-vous nous permettaient de mettre au clair les objectifs, ainsi que d'autres idées évoquées sur le compte rendu rédigé la veille. De plus, c'est durant ces rendez-vous que l'on discutait des problèmes rencontrés et que l'on cherchait ensemble les solutions.

Ainsi, durant le tout premier rendez-vous, nous avons décidé de réfléchir à la structure principale de notre projet. Quelles classes créer, quelles fonctions chacune de ces classes posséderait, quelles classes hériteraient des autres. Enfin, la première répartition des tâches s'est faite arbitrairement durant ce premier rdv, chacun choisissant une classe ou deux à implémenter.



Les différentes classes, attributs et fonctions décidées à l'issue du premier rdv

Nous avons bien entendu créé un groupe WhatsApp et avons tous lié nos environnement de travail à GitLab.

Et ainsi, avec des objectifs et une répartition des tâches bien définis pour la semaine qui suit le rdv, nous avons commencé à développer le projet.

Cahier des charges :

Une des particularités de notre projet est la rédaction d'un cahier des charges « dynamique ». Effectivement, il a été décidé dès le début qu'un cahier des charges serait rédigé et entretenu au fur et à mesure de la progression afin de nous guider sur les fonctionnalités à programmer à l'avenir. Il était ainsi mis à jour, dès qu'une nouvelle idée était retenue, avec l'ajout des consignes pour la programmation de la dite fonctionnalité/idée. Concrètement, ce cahier des charges contient toutes les charges minimales imposées par le sujet donné par l'UE, mais également toutes les charges supplémentaires que le groupe à décider d'implémenter : **les extensions** (que l'on abordera par la suite dans e rapport), ainsi que les consignes sur comment les programmer dans les grandes lignes. Enfin, le cahier des charges contient également des informations sur les différents sorts implémenter, ainsi qu'une maquette servant pour l'interface de l'extension principale de notre jeu : **Héros Personnalisable**.

Diagramme des classes :

Nous avons également décidé de créer un diagramme des classes, afin de nous guider au mieux dans l'organisation de ces dernières. Ce dernier se trouve sur le Git. Version simplifiée dans l'annexe à la fin.

Organisation des classes :

Notre code suit plus ou moins le modèle MVC (Modèle, Vue, Contrôleur). Ainsi, le package **Graphics** contient les classes liées à l'interface graphique, le package **main** contient les classes qui vont gérer le modèle et celles qui vont gérer le contrôleur. Cependant, certaines classes peuvent remplir plusieurs fonctions à la fois. On peut citer notamment **Playing** qui gère l'interface graphique de plusieurs éléments présents à l'écran, mais est également la classe qui sert de pivot lors d'une partie. C'est une classe qui sert de point relais entre toutes les autres classes.

Modèle de jeu :

Entités - Héros :

Nous restons sur le modèle du jeu classique. Soit cinq héros, chacun d'une classe différente, et chacun possédant 4 améliorations uniques, et donc 5 dés différents chacun. Autant dire que les possibilités pour une formation à n'importe quel moment de la partie peuvent énormément varier.

Voici les 5 héros de base de notre jeu :



Archer

Épéiste

Tank

Mage

Guérisseur

À noter que chacun des sprites des héros, sprites des héros améliorés compris, ont été fait à la main par nous-mêmes. Ce sont des créations originales.

Du côté du programme, chacun de ces héros possède sa propre classe (classe java ici, pas la classe du héros en tant que telle) du même nom qui héritent toutes d'une classe **Heros** elle-même héritant de la classe abstraite **Entites**. Faisant partie des premières classes définies au niveau de la chronologie du projet, il nous a paru logique, puisque nous allions avoir deux types d'entités, soit les Héros et les Ennemis, de créer une classe mère Entites contenant tous les attributs et fonctions que possèdent en commun les deux types d'entités. C'est de ce fait que la classe Heros hérite de la classe Entites (les points de vie et autre attributs communs avec les entités Ennemis). Mais notre jeu comportant 5 classes de héros différentes, nous avons préféré séparer ces cinq classes en cinq classes Java, ce qui nous permet plus facilement d'utiliser les différents types de héros sans devoir passer par une vérification du nom, par exemple. Cette méthode facilite également l'instanciation de chaque type de héros, puisqu'un simple appel du constructeur suffit. Enfin, l'une des raisons principales pour lesquelles nous avons opté pour cette organisation des héros, c'est notre **extension Améliorations**, qui implique des fonctions distinctes entre chaque type de héros, chose que l'on verra plus tard dans ce compte rendu.

Entités - Ennemis :

Les ennemis de base, contrairement aux héros, sont bien plus nombreux. Ils sont également divisés en plusieurs catégories soit :

- **Sbire** : ce sont les ennemis de base, les plus faibles
- **Capitaines** : ce sont des ennemis qui tapent plus fort que les sbires
- **Sous-boss** : ce sont les ennemis que l'on retrouve avant d'affronter les boss

- **Boss** : les gardiens de milieu et fin de niveau, ce sont les ennemis les plus puissants
- **Boss Final** : ennemi le plus puissant du jeu, il n'apparaît qu'à la fin du niveau final

Ainsi, chacun des ces types d'ennemis possèdent plusieurs ennemis, pour un total de 24 ennemis distincts. Ce nombre étant beaucoup trop grand pour des classes distinctes, nous avons opter pour une autre méthode : ce sont les sous types d'ennemis qui possèdent des classes attitrées. La classe **Sbire**, **Capitaine**, **SousBoss**, **Boss** et **BossFinal** héritent toutes de la classe **Ennemi**, elle-même héritante de la classe **Entites**, comme mentionné précédemment. Cependant, pour différencier chaque ennemi, chacune de ces classes possèdent un attribut **nomsPossibles** qui est un tableau de String contenant tous les noms possibles pour cette classe d'ennemis, soit tous les ennemis distincts de cette classe d'ennemis. Ci-dessous un exemple :

```
String [] nomPossibles = {"Amalgame Noire", "Moustique", "Oie", "Taureau", "Crapeau", "Squelette", "Loup-Garou", "Sbire"}
```

Attribut nomsPossibles de la classe Sbire

```
public void attribuerType(Sbire s){ 1 usage  ↳ Kai WITZMANN +2
    Random rd = new Random();
    int randomNumber = rd.nextInt()%nomPossibles.length;
    randomNumber = Math.abs(randomNumber);
    s.setNomEntite(nomPossibles[randomNumber]);
    s.setNomImage(nomPossibles[randomNumber].replace(oldChar: ' ', newChar: '_'));
    s.setDe(importeDeSbire(nomPossibles[randomNumber]));
}
```

Fonction attribuerType de la classe Sbire

Cette fonction, présente dans toutes les sous classes de Ennemis, prend en pratique une instance vide de Sbire en paramètre et va remplir cette dernière en générant dans un premier temps un nombre aléatoire à partir duquel elle attribue son nom d'ennemi à cette instance. C'est par ce processus que le Sbire (et les ennemis en général) obtient son identité. Puis, il y a le remplissage du dé de cet ennemi nouvellement créé.

```

protected static De importeDeSbire(String nom){ //sert à créer le dé du Sbire : reste à voir les attaques possibles. 2 usages ▾ GabrielFischer +1
    if (nom==null){
        switch (nom){
            case "Amalgame Noire" : De deAmalgame = new De(new Fear(niveau: 2),new Griffures(niveau: 2),new Griffures(niveau: 2),new Blindness(niveau: 2), null, null);
            return deAmalgame;
            case "Moustique" : De deMoustique = new De(new Confusion(niveau: 1),new Blindness(niveau: 2),null,new Confusion(niveau: 2), new Blindness(niveau: 2), null);
            return deMoustique;
            case "Die" : De deDie = new De(new Goose_Pinch(niveau: 2),new Goose_Pinch(niveau: 1),new Griffures(niveau: 2),new Griffures(niveau: 1), new Confusion(niveau: 2), null);
            return deDie;
            case "Taureau" : De deTaureau = new De(new Destruction(niveau: 3),new Blindness(niveau: 1),new Blindness(niveau: 1),new Fear(niveau: 2), new Fear(niveau: 2),null);
            return deTaureau;
            case "Crapaud" : De deCrapaud = new De(new Fear(niveau: 2),new Blindness(niveau: 2),new Blindness(niveau: 1),new Destruction(niveau: 2), null, null);
            return deCrapaud;
            case "Squelette" : De deSquelette = new De(new Confusion(niveau: 2),new Destruction(niveau: 2),new Fear(niveau: 1),new Griffures(niveau: 2), new Fear(niveau: 2), null);
            return deSquelette;
            case "Loup-Garou" : De deLoupGarou = new De(new Griffures(niveau: 2),new Griffures(niveau: 2),new Fear(niveau: 2),new Fear(niveau: 1), new Griffures(niveau: 1), null);
            return deLoupGarou;
            default: De deSbire = new De(new Griffures(niveau: 2),new Griffures(niveau: 2),null,new Griffures(niveau: 2), new Griffures(niveau: 2), null);
            return deSbire;
        }
    }else{
        De deSbire = new De(new Griffures(niveau: 2),new Griffures(niveau: 1),new Griffures(niveau: 1),new Griffures(niveau: 1), new Griffures(niveau: 1), new Griffures(niveau: 1));
        return deSbire;
    } 5 usages ▾ Kai WITZMANN +1
}

```

Fonction importeDeSbire de la classe Sbire

C'est cette fonction, également présente chez toutes les sous classes de Ennemis, qui va créer et retourner le dé correspondant au nom de l'instance de Sbire en cours de création.

C'est donc de cette manière que l'on a réussi à garder une bonne diversité d'ennemis, tout en évitant le surplus de classes inutile chez l'ensemble des entités.

Or, pour pouvoir profiter de ces entités, il faut pouvoir jouer.

Lancement de la partie :

Le Joueur :

Le jeu possède une classe Joueur, correspondant bien au joueur. Elle contient toutes les informations nécessaires au bon fonctionnement de la partie, notamment en ce qui concerne directement les ressources du joueur. Ainsi, elle contient les deux types de monnaie que peut posséder le joueur, mais également la liste des héros qui lui sont disponibles avant le lancement d'une partie, et enfin les sorts qui lui sont également disponibles.

Choix de difficulté :

Notre jeu permet de sélectionner un mode de difficulté avant chaque partie parmi 4 niveaux de difficulté : Facile, Normal, Difficile et Inferno. Naturellement, les ennemis sont plus coriaces plus la difficulté augmente. La difficulté Inferno quant à elle octroie pleine puissance aux ennemis considérés comme faibles, et les ennemis auparavant considérés comme forts sont rendus faible, rendant ainsi la partie, comme son nom l'indique, infernale.

Déroulement d'une partie :

Une fois la difficulté choisie, le jeu génère des ennemis aléatoirement en adéquation avec cette dernière. Cette génération aléatoire des ennemis se fait via une fonction **genererEnnemis**, présente dans la classe (Test)GenerationEnnemi du package main, dont le but est de vérifier la progression du joueur dans la partie afin de générer les bons types d'ennemis. Une fois les ennemis générés, le jeu choisit aléatoirement les capacités de ces derniers. Ainsi, comme dans le jeu de base, le joueur a la possibilité de voir à l'avance non seulement quelle capacité les ennemis vont utiliser, mais il pourra également voir quel héros va se faire attaquer et combien de dégâts ce-dernier va subir. Ci-dessous un exemple :



Capture d'écran du jeu, début du tout premier tour de jeu :

○ : les capacités des ennemis

- - - : montre quel héros est visé par l'ennemi sur lequel le joueur place le curseur

-> : indique le nombre de points de vie que le héros va perdre

Choix des capacités - Système de dés :

La particularité de Slice & Dice est que le joueur ne contrôle pas totalement les actions de ses héros. En effet, le gameplay ici est le suivant : à chaque début de tour, le joueur peut lancer jusqu'à trois fois des dés. Ces dés, qui sont au nombre de héros, contiennent naturellement 6 faces chacun, chacune d'entre elles contenant une capacité. Le joueur peut donc lancer les dés, mais il a aussi la possibilité

de sélectionner une ou plusieurs capacités avant de relancer les dés. Dans ce dernier cas, seuls les dés dont la face n'a pas été choisie seront relancés.

Choix des capacités - Capacités (Héros) :

Dans notre jeu, une capacité utilisée par un héros possède un niveau. Ce niveau détermine la puissance de la capacité, ou plutôt son efficacité. En général donc, plus le niveau d'une capacité est élevé, mieux elle est. Ce niveau est indiqué en bas à droite de la face (du dé) contenant l'icône de la capacité. Il existe 5 capacités de base pour les héros :



Arc

Bouclier

Épée

Mana

Soin

L'Arc est une capacité qui inflige des dégâts et dont la portée lui permet d'atteindre des cibles en retrait. Plus son niveau augmente, plus elle inflige de dégâts.

Le Bouclier est une capacité qui octroie de la protection au héros sur lequel est appliquée la capacité. Plus son niveau augmente, et plus elle octroie du bouclier.

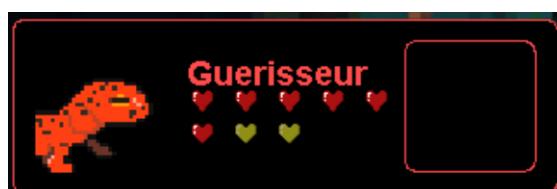
L'Épée est une capacité qui inflige des dégâts. Plus son niveau augmente, plus elle inflige de dégâts.

Le Mana est une capacité qui octroie du mana. Plus son niveau augmente, plus elle octroie du mana.

Le Soin est une capacité qui régénère les points de vie du héros sur lequel elle est appliquée. Plus son niveau augmente, plus le nombre de points de vie restaurés sont nombreux. (*pour le mécanisme p.17)

Choix des capacités - Capacités (Ennemis) :

Le choix des capacités chez les ennemis se fait aléatoirement par le jeu au début de chaque tour, comme dans le jeu original. Cela permet au joueur d'établir une stratégie en fonction des actions ennemis imminentes.



Les coeurs jaunes indiquent les points de vie qui vont être perdus à la fin du tour

Seulement des capacités pour gagner ?

Bien sûr que non ! Notre jeu propose diverses manières d'aider le joueur afin de gagner la partie.

Équipements :

Les équipements peuvent logiquement être équipés durant la partie, sont au nombre 8 et héritent tous de la classe mère Equipement. Les effets de ces équipements s'appliquent grâce à la fonction override effet() présente dans Equipement et dont les corps distincts correspondants aux différents équipements sont implémentés dans ces différentes sous classes.

Choose Implementation of Equipement.effet() (8 found)		
 BouclierDivin	(Equipement)	slice-and-dice
 Faucon	(Equipement)	slice-and-dice
 ForceGauche	(Equipement)	slice-and-dice
 GanteletsDePuissance	(Equipement)	slice-and-dice
 Pansements	(Equipement)	slice-and-dice
 PetitCoeur	(Equipement)	slice-and-dice
 Ricochets	(Equipement)	slice-and-dice
 RivièreDeMana	(Equipement)	slice-and-dice

Les différentes classes d'équipements implémentant la fonction effet

```
public void effet(){ ▲ GabrielFischer +1
    if (perso!=null){
        for (Face face : perso.getDe().face){
            if(face.attaque!=null){
                if (face.attaque.getType()==3||face.attaque.getType()==13||face.attaque.getType()==23||face.attaque.getType()==34){
                    face.attaque.setNiveau(face.attaque.getNiveau()+1);
                }
            }
        }
    }
}
```

Fonction effet() implémentée dans la classe BouclierDivin.java : vérifie si l'équipement est attribué à un héros, puis augmente de 1 niveau chaque capacité de type Bouclier présent sur le dé du héros



Interface de l'inventaire, on notera que chaque héros ne peut posséder que jusqu'à 2 équipements à la fois pour éviter que ce ne soit trop facile

Cependant, ces équipements ne sont pas donnés de la même manière que dans le jeu de base. C'est ici qu'entre en jeu notre première **extension : la boutique d'équipements.**

Extension - Boutique d'équipements :

La boutique d'équipements est, comme son nom l'indique, un onglet dans lequel les équipements peuvent être achetés.



Le joueur accède à la boutique d'équipements via cette icône

La boutique d'équipements n'est cependant pas une simple boutique. Nous voulions rendre l'obtention des équipements plus amusante et donc, nous avons décidé de programmer la boutique comme une loterie. En effet, la boutique générera à chaque début de tour trois équipements différents, parmi lesquels le joueur pourra acheter ce dont il a besoin en **1 seul exemplaire à la fois.**

Une fois les équipements achetés, ces derniers sont stockés dans l'onglet Inventaire, onglet qui permet au joueur de voir quels équipements il a en stock mais également de les équiper à ses héros.



Interface de la boutique; **les sprites des équipements sont originaux**

Une boutique implique la présence d'une monnaie. Cette monnaie est stockée dans l'attribut monnaiePartie de la classe Joueur, et est incrémentée à chaque ennemi vaincu. En effet, chaque classe d'ennemi possède une fonction monnaieLachee() qui permet au programme de vérifier quel ennemi vient d'être vaincu, et d'octroyer la monnaie correspondante à ce dernier au joueur. La quantité de monnaie gagnée dépend bien sûr de la difficulté de l'ennemi vaincu. Autrement dit, si le joueur veut acheter des équipements, il doit battre des ennemis.

```
public int monnaieLachee(){ 5 usages ▾ Kai WITZ
    switch (this.getNomEntite()){
        case "Amalgame Noire" : return 100;
        case "Moustique" : return 150;
        case "Oie" : return 75;
        case "Taureau" : return 50;
        case "Crapeau" : return 50;
        case "Squelette" : return 50;
        case "Loup-Garou" : return 50;
        default: return 70;
    }
}
```

Fonction monnaieLachee() de la classe Sbire

Sorts :

Les sorts, présents également dans le jeu original, sont des capacités magiques que le joueur peut utiliser s'il possède assez de mana. Dans notre jeu, les sorts sont au nombre de 10, et héritent de la classe CapaciteSpe, qui hérite elle-même de la classe Capacite. Tout comme dans le jeu de base, seuls les héros de la classe Guerisseur et Mage influent sur la disponibilité des sorts. Ces deux héros, ainsi que chacune de leurs améliorations possèdent des sorts exclusifs.



Les sorts disponibles, au nombre de 3, sont affichés en bas de l'écran



Le sort **Renew**, appartenant au **Guerisseur**, a été remplacé par le sort **Infuse** lorsque le guerisseur a été amélioré en **Apothicaire**

Le sort Burst est quant à lui indépendant. Il ne dépend donc d'aucun héros et peut par conséquent être utilisé par le joueur à tout moment, sous réserve que ce dernier possède assez de mana. Les sprites utilisés pour les sorts sont également tous originaux et ont été faits par nous-mêmes.

Les effets des sorts sont divers, Burst et Flare sont des sorts qui infligent des dégâts à un ennemi, mais Renew est un sort qui soigne un héros. D'autres sorts comme Drop par exemple ont des effets plus complexes : Drop inflige des dégâts seulement à l'ennemi qui possèdent le plus de points de vie. Scald est un sort qui va infliger des dégâts à tous les ennemis qui ont déjà subi des dégâts, etc.

Le programme va faire appel à plusieurs fonctions pour assurer le bon fonctionnement des sorts. Dans la classe Jeu, qui gère la partie, la fonction attaqueSpe() sert à vérifier que le sort est utilisé une fois que tous les lancers de dés ont été effectués, et si le joueur clique sur un sort, alors l'attribut briqueAttaqueMana prend la valeur de la brique du sort correspondant. Puis, la fonction choixSort() est appelée. Cette fonction sert de rond-point, au sens que c'est dans cette fonction que le programme vérifie sur quel sort exactement le clic a été effectué.

```
public boolean attaqueSpe(){ 3 usages ↗ GabrielFischer
    if(playing.getToolbar().rollDone()){
        if(brickAttaqueMana!=null && !brickAttaqueMana.getUsed()){
            if(choixSort()){
                brickAttaqueMana.setUsed(true);
                return true;
            }
        }
    }
    return false;
}
```

Fonction attaqueSpe() de la classe Jeu

```
public boolean choixSort(){ 1 usage  ↗ GabrielFischer
    if(brickAttaqueMana!=null && brickAttaqueMana.getMana()!=null && brickAttaqueMana.getMana().getNom()!=null){
        String name = brickAttaqueMana.getMana().getNom();
        switch (name){
            case "Balance" : sortBalance(playing.getListeEnnemis(), playing.getListeHeros()); return true;
            case "Bind" : sortBind(playing.getListeEnnemis(), playing.getListeHeros()); return true;
            case "Blaze" : if(ennemisSelected!=null){
                sortBlaze(ennemisSelected);
                return true;
            } return false;
            case "Burst" : if(ennemisSelected!=null){
                System.out.println("test2");
                sortBurst(ennemisSelected);
                System.out.println("test3");
                return true;
            } return false;
            case "Drop" : sortDrop(playing.getListeEnnemis()); return true;
            case "Flare" : if(ennemisSelected!=null){
                sortFlare(ennemisSelected);
                return true;
            } return false;
            case "Infuse" : sortInfuse(playing.getListeHeros()); return true;
            case "Liquor" : sortLiquor(playing.getListeHeros()); return true;
            case "Renew" : if(heroSelected!=null){
                sortRenew(heroSelected);
                return true;
            } return false;
        }
    }
}
```

Une partie de la fonction choixSort() de la classe Jeu

La fonction choixSort() vérifie donc le nom du sort et appelle la fonction qui va appliquer l'effet du sort correspondant. Ces fonctions sont présentes dans la même classe Jeu et sont toutes sous le format sortNomDuSort().

Les fonctions de ce type procèdent toutes de la même manière dans le sens où elles créent une instance temporaire du sort en question et appellent la fonction action du sort correspondant, puis enlèvent au joueur le nombre de mana qu'il faut.

```
public void sortFlare(Ennemis ennemiSelectionne) 1 usage  ↗ Johnson Michael
{
    Flare tmp = new Flare();
    tmp.action(ennemiSelectionne);
    this.joueur.setMana(this.joueur.getMana()-tmp.consommationMana);
}
```

Fonction sortFlare() de la classe Jeu

```

public void sortDrop(Ennemis[] listeDesEnnemisSurLeTerrain)//cette f
{
    Drop tmp = new Drop();
    Ennemis ennemiTmp = listeDesEnnemisSurLeTerrain[0];
    for(Ennemis enn : listeDesEnnemisSurLeTerrain)
    {
        if (enn.getPdv() > ennemiTmp.getPdv())
        {
            ennemiTmp=enn;
        }
    }

    tmp.action(ennemiTmp);
    this.joueur.setMana(this.joueur.getMana()-tmp.consommationMana);
}

```

Fonction sortDrop(), autre exemple avec cette fois un sort qui inflige des dégâts uniquement à l'ennemi possédant le plus de pdv

Les fonctions action() qui sont appelées dans chacune de ces fonctions correspondent aux fonctions overrides présentes dans les classes Capacite -> CapaciteSpe -> SortSpécifique. Si on prend l'exemple du sort Drop, alors la fonction sortDrop finit par appeler la fonction action avec en argument un Ennemi. Or, ce premier appel se fait à partir de la fonction override action présente dans Capacite :

```

public void action(Heros h){} 30 overrides
public void action(Ennemis e){ 12 overrides

```

Deux fonctions overrides action de la classe Capacite

Ces fonctions étant des overrides, et l'appel se faisant à partir d'une instance de Capacite.CapaciteSpe.Drop, le programme va aller chercher la fonction dans la classe CapaciteSpe :

```

public void action(Ennemis e){} 7 overrides
public void action(Heros e){} 6 overrides

```

Deux fonctions override action de la classe CapaciteSpe

Enfin, l'appel se poursuit jusqu'à atteindre la fonction action présente dans chacune des classes de chaque sort.

```

public void action(Ennemis e)
{
    this.attaque(e);
    /*cette fonction ne prend
     * le plus de points de vie
}

```

Fonction action de la classe Drop, elle appelle la fonction attaque (qui enlève des pdv a e) de la classe Capacite

*On notera que le mécanisme d'appels de fonctions overrides est le même pour les capacités basiques de notre jeu.

Extension - Poison :

Le poison est une fonctionnalité que l'on a décidé d'attribuer exclusivement aux héros, puisque cela pourrait aider contre les boss qui ont énormément de points de vie. La capacité Poison possède sa propre classe, qui hérite de Capacite, cependant son fonctionnement diffère. En effet, là où pour les autres capacités, le niveau détermine l'efficacité de la dite capacité, le niveau de la capacité Poison indique plutôt le nombre de tours durant lequel l'ennemi va subir le poison. En effet, le poison inflige toujours 2 points de vie et ce, tant que l'attribut estEmpoisonné de l'ennemi est supérieur à 0. Le poison s'applique à la fin de chaque tour. Pour cela, la fonction applicationDuPoison() est appelée à chaque fin de tour.

```

public void applicationDuPoison(Ennemis [] ennemisSurLeTerrain)
    /*cette fonction doit être appelée à chaque fin de tour,
     elle sert à appliquer le poison aux ennemis concernés IMPORTANT
     d'un tableau vers une liste*/
{
    for(Ennemis e : ennemisSurLeTerrain)
    {
        if(e.getEstEmpoisonne()>0)
        {
            if(e.getPdv()-2<=0)
            {
                e.setPdv(0);
                e.setEstEmpoisonne(0);
            }
            else
            {
                e.setPdv(e.getPdv()-2);
                e.setEstEmpoisonne(e.getEstEmpoisonne()-1);
            }
        }
    }
}

```

Extension - Évènements aléatoires :

Les évènements aléatoires sont également un moyen pour le joueur de profiter d'une occasion de reprendre l'avantage, ou bien de le perdre. En effet, ces évènements aléatoires sont des évènements qui ont 1 chance sur 24 de se produire à chaque début de vague. Ils sont au nombre de trois et peuvent permettre au joueur de prendre soit un bonus soit un malus.

```
public void eventAleatoire(Heros [] listeHerosSurLeTerrain)
{
    Random generateur = new Random();
    int nbAlea = generateur.nextInt( bound: 25 );
    if(nbAlea == 16)
    {
        choixDeLeventAleatoire(listeHerosSurLeTerrain);
    }
}
```

Fonction eventAleatoire() de la classe Jeu

Cette fonction permet de déterminer si un event aléatoire aura lieu ou non au début de ce tour.

```
public void choixDeLeventAleatoire(Heros[] listeHerosSurLeTerrain)
{
    Random generateur = new Random();
    int nbAlea = generateur.nextInt( bound: 3 );
    if(nbAlea==0)
    {
        eventEnleveDesPdv(listeHerosSurLeTerrain);
    }
    else if(nbAlea==1)
    {
        eventBouclierPourTous(listeHerosSurLeTerrain);
    }
    else
    {
        eventAjoutMana();
    }
}
```

Fonction choixDeLeventAleatoire() de la classe Jeu

Cette fonction permet de choisir l'event aléatoire qui aura lieu. Parmi eux se trouve l'ajout de mana, l'ajout de bouclier à tous les héros et enfin le vol de points de vie chez tous les héros.

Améliorations :

Chaque héros possède deux lignées d'améliorations possibles, pour un **total donc de 20 types de héros**. Pour rendre le jeu plus compliqué, nous avons décidé de changer le processus d'amélioration. En effet, notre jeu propose aléatoirement un héros à améliorer à la fin du niveau. Le joueur est **obligé** de choisir une proposition parmi les deux proposées. Ce faisant, le héros est amélioré et passe au niveau 2. Si un héros au niveau 2 est proposé par le jeu pour être amélioré, alors **seul l'amélioration correspondant à sa lignée** est proposée, et le joueur est obligé de l'améliorer au niveau 3. Cela force le joueur à adopter de nouvelles stratégies et permet de varier un minimum le gameplay. Prenons l'exemple du Mage. Ce dernier possède la première lignée d'amélioration Mage->Sorcier->Chaman et la deuxième lignée d'amélioration Mage->Arcaniste->Demoniste.



Interface d'amélioration pour le Mage

Le Mage est au niveau 1, donc le jeu propose deux choix, soit l'amélioration en Sorcier, soit l'amélioration en Arcaniste.

Si le joueur choisit le Sorcier, le Mage est amélioré en Sorcier, idem pour l'Arcaniste. Si le jeu sélectionne la classe Mage pour être améliorée de nouveau, alors cette fois la seule proposition qui sera affichée sera le Chaman, puisque ce dernier étant l'amélioration de niveau 3 du Sorcier. Si par contre le joueur avait choisi Arcaniste, ce serait le Demoniste qui serait proposé.

Ici, le jeu a re sélectionné la classe Mage pour être améliorée, mais comme le Sorcier a été choisi comme amélioration au niveau 2, c'est le Chaman qui est proposé.



Interface d'amélioration pour le Sorcier

Plusieurs fonctions sont appelées pour permettre le bon fonctionnement des améliorations. La toute première fonction se trouve dans la classe Playing qui, on le rappelle, est une classe qui gère l'interface graphique. La fonction initBrickAmelioration() de Playing permet l'initialisation de l'interface d'amélioration qui sera proposée au joueur à la fin du tour. La fonction handleAmeliorationClick de la classe Playing va être la fonction qui va gérer l'amélioration en fonction de sur quelle amélioration le joueur a cliqué.

```
private void quelleAmeliorationChoisie(BrickAmelioration briqueChoisie, int i)
{
    if(this.combat.verifAmeliorationLigneMage(briqueChoisie, i))
    {
        return;
    }
    else if(this.combat.verifAmeliorationLigneGuerisseur(briqueChoisie, i))
    {
        return;
    }
    else if(this.combat.verifAmeliorationLigneArcher(briqueChoisie, i))
    {
        return;
    }
    else if(this.combat.verifAmeliorationLigneEpeiste(briqueChoisie, i))
    {
        return;
    }
    else if(this.combat.verifAmeliorationLigneTank(briqueChoisie, i))
    {
        return;
    }
}
```

La fonction quelleAmeliorationChoisie est une fonction intermédiaire qui va permettre d'identifier la classe du héros à améliorer, puis va appeler la fonction de vérification de l'amélioration de la lignée du héros correspondant.

```
public boolean verifAmeliorationLigneetank(BriqueAmelioration briqueChoisie, int i) 1 usage ▲ Johnson Michael
{
    if(briqueChoisie.getHero().getNomEntite().equals("Tank") && i==0)
    {
        for(int k=0; k<this.playing.getBrickHeroes().length; k++)
        {
            if(this.playing.getBrickHeroes()[k].getHero().getNomEntite().equals("Tank"))
            {
                this.playing.getBrickHeroes()[k].getHero().ameliorationAuNiveau2(ameliorationChoisie: "Colosse");
                return true;//amélioration en brute réussie
            }
        }
    }
    else if(briqueChoisie.getHero().getNomEntite().equals("Tank") && i==1)
    {
        for(int k=0; k<this.playing.getBrickHeroes().length; k++)
        {
            if(this.playing.getBrickHeroes()[k].getHero().getNomEntite().equals("Tank"))
            {
                this.playing.getBrickHeroes()[k].getHero().ameliorationAuNiveau2(ameliorationChoisie: "Brute");
                return true;//amélioration en colosse réussie
            }
        }
    }
}
```

Une partie de la fonction verifAmeliorationLigneetank de la classe Jeu

Les captures d'écran ont été effectuées avant certaines modifications de dernière minute, il est donc normal si la syntaxe n'est pas la même que dans le code fourni sur le git. En cas de doute, se référer au code fourni dur le git.

Poursuivons avec l'exemple du Tank :

```
public void ameliorationAuNiveau2(String ameliorationChoisie)
{
    if(ameliorationChoisie.equals("Brute"))
    {
        this.ameliorationNiv2Brute();
    }
    else if(ameliorationChoisie.equals("Colosse"))
    {
        this.ameliorationNiv2Colosse();
    }
}

//Premiere proposition si le héros est au niveau 1
public void ameliorationNiv2Brute() 1 usage ▲ Johnson Michael
{
    this.setPdvMax(this.getPdvMax()+4);
    this.setPdv(this.getPdvMax());
    this.setDe(this.getDeAmeliorationUnDeux());
    this.setNomEntite("Brute");
    this.incrementationNiveauActuel();
}
```

Il y a une fonction de ce type par classe de héros de base, soit cinq fonctions au total, cela permet de mieux séparer les classes et de ne pas avoir une fonction de plus de 200 lignes. Ces fonctions ont pour but de vérifier à quel niveau d'amélioration le héros sélectionné est, et d'appeler la fonction d'amélioration correspondante. Les fonctions d'améliorations sont des fonctions d'amélioration override présentes en premier lieu dans la classe Heros, puis présente dans chacune des classes des cinq classes avec un corps distinct dans ces dernières pour permettre les 20 améliorations possibles.

```
public void ameliorationAuNiveau2(String ameliorationChoisie){}

public void ameliorationAuNiveau3() { 11 usages 5 overrides • Johnson N
}
```

Deux fonctions d'amélioration override de la classe Heros

Ce mécanisme peut paraître complexe de prime abord, mais il permet de bien distinguer les différents héros, ainsi que leurs lignées et permet par conséquent au programme d'appliquer le concept d'amélioration concrètement sans se tromper. On notera enfin que la fonction `remplacementSortGuerisseur/Mage()` est appelée uniquement si l'amélioration se fait sur la classe Mage ou Guérisseur afin de remplacer le sort du précédent héros par le nouveau.

Nous vous laissons découvrir toutes les améliorations possibles en jouant au jeu !

Et après une vague ?

Une fois la première vague vaincue, vous êtes partis pour 11 vagues supplémentaires, avec plusieurs types d'ennemis qui vous attendront, notamment le boss final au dernier niveau. Cela fait donc un total de 12 vagues différentes, chacune complètement aléatoire, excepté la présence ou nom de certaines classes d'ennemis bien sûr. **Vous remarquerez également le changement des arrières plans lorsque le joueur avance dans les vagues, une extension que l'on a décidé d'implémenter pour améliorer l'esthétisme de notre jeu.**

Differents menus :

Menu Principal :

Bien sûr, la première chose devant laquelle le joueur se retrouve lorsqu'il lance le jeu, c'est bien le menu principal :



Menu principal

C'est à partir de ce menu que le joueur peut décider de lancer une partie en cliquant sur la difficulté de son choix. Mais ce menu principal donne également accès à d'autres menus accessibles via les icônes en haut à droite.

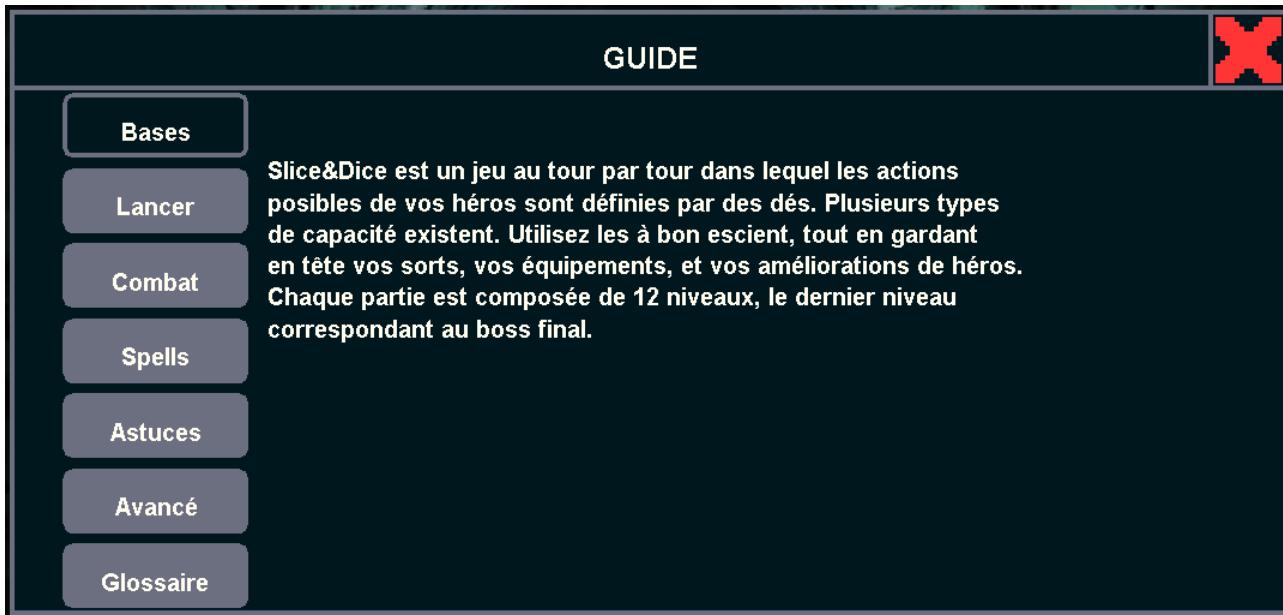
Paramètres :



Menu des paramètres

Seul le paramètre « Volume » est ajustable pour notre jeu. Ce menu permet au joueur de baisser, augmenter, ou mute le son.

Guide :



Section « Bases » du menu Guide

Le menu guide est une aide au joueur débutant. Elle est dotée de plusieurs sections, chacune contribuant à améliorer la prise en main du concept du jeu.



Menu Succès

Extension - Boutique de capacités :



Boutique de capacités

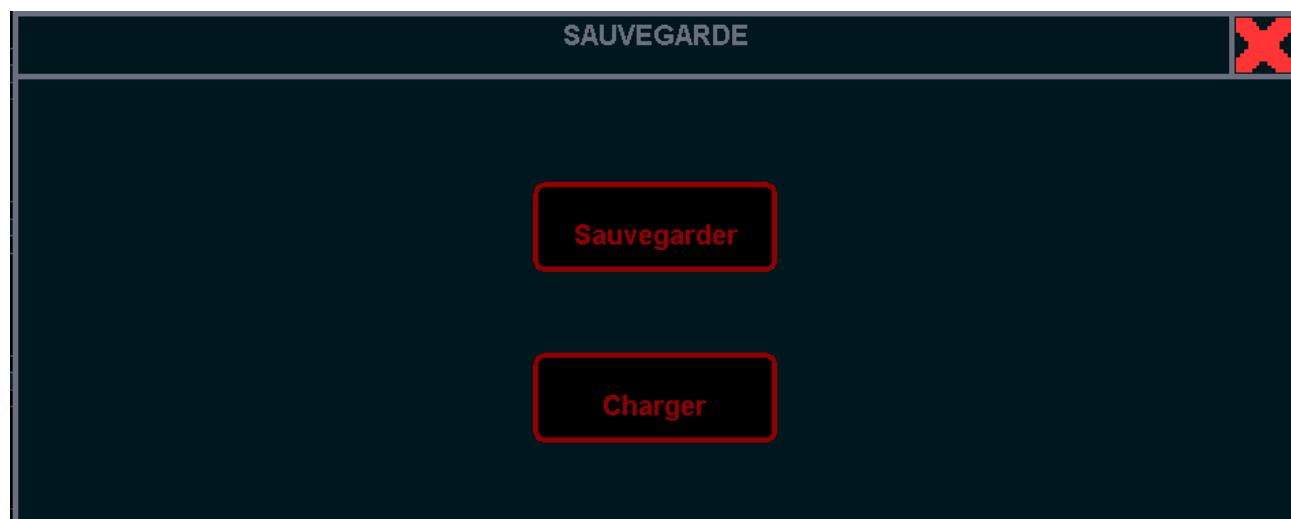
À ne pas confondre avec la boutique d'équipements accessible pendant une partie, la **boutique de capacités** permet au joueur d'**acheter des capacités supplémentaires** qui lui sera possible de donner à son héros personnalisé (extension Custom). Les capacités présentes dans cette boutique sont des capacités qui ont toutes un effet particulier, souvent des double effets.

Extension - Succès :

Notre jeu possède une autre **extension** parmi les menus, celle des **succès**. En effet, pour rendre le jeu plus rejouable, nous avons décidé d'implémenter un système de succès qui récompense le joueur en « Gold » lorsqu'il remplit l'une des tâches mentionnées dans le menu Succès.

Extension - Sauvegarde :

En effet, notre jeu implémente l'extension de sauvegarde. Il s'agit dans notre cas d'une sauvegarde automatique, c'est-à-dire que lorsque le jeu se lance, un profil Joueur est créé et est stocké dans un fichier joueur.dot. De même, le joueur est capable de sauvegarder en pleine partie. Signifiant donc que lorsque le joueur quitte le jeu, relance le jeu et une partie et clique sur la recharge d'une sauvegarde, la partie sauvegardée est rechargée et peut être reprise là où elle s'était arrêtée.



Menu de Sauvegarde ; Cliquer sur « Sauvegarder » permet de sauvegarder la partie dans le fichier save.dot; Cliquer sur « Charger » permet de recharger la partie à partir du fichier save.dot

Extension - Custom :

L'extension Custom (ou héros personnalisable) est notre « extension principale » si l'on doit en citer une. Grâce à cette extension, le joueur peut créer son propre héros, en plus de ceux déjà existants.

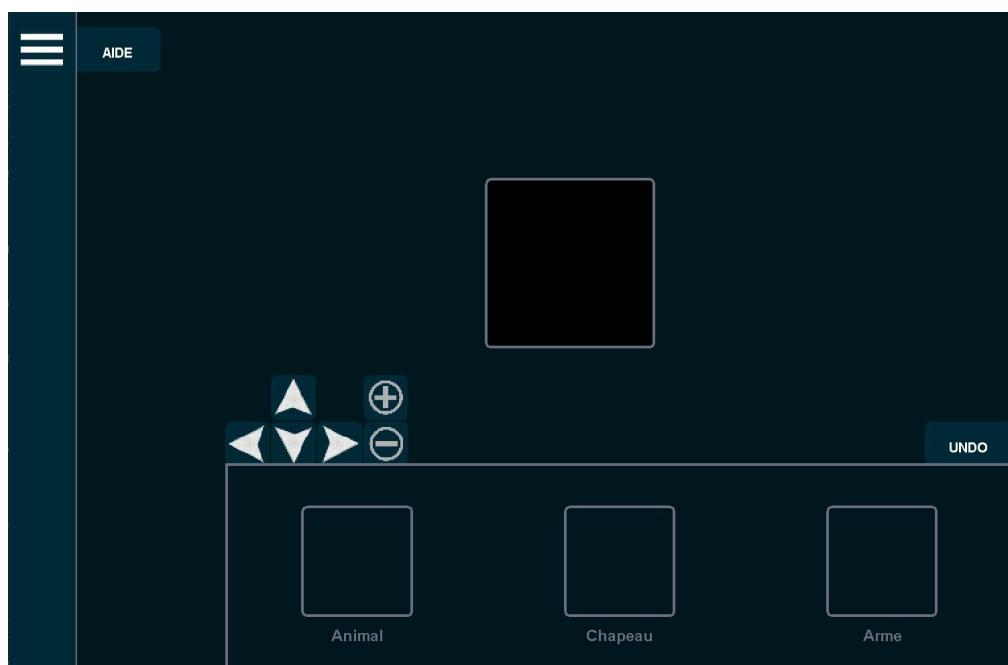
Les captures d'écran ont été faites avant quelques modifications post-rédactionnelles.

Les classes gérant Custom sont toutes présentes dans le package Graphics, puisqu'il s'agit dans un premier temps de permettre au joueur de personnaliser le visuel de son héros, donc manipulation visuelle.

Partie visuelle :

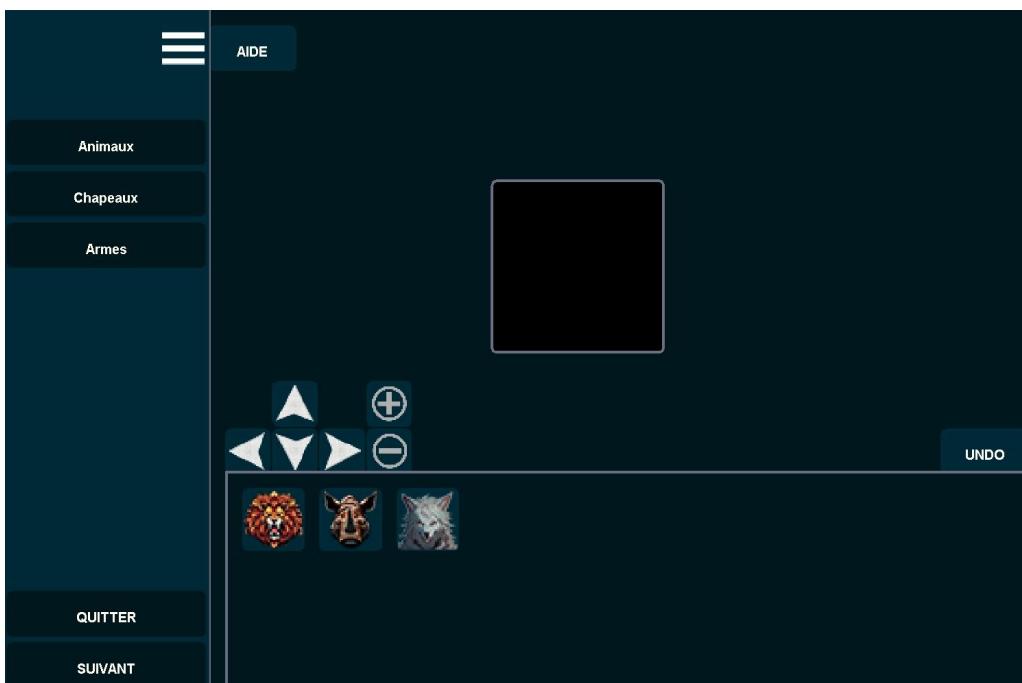


Icône présente en haut du menu principal et donnant accès à l'extension Custom



Page d'accueil du menu Custom

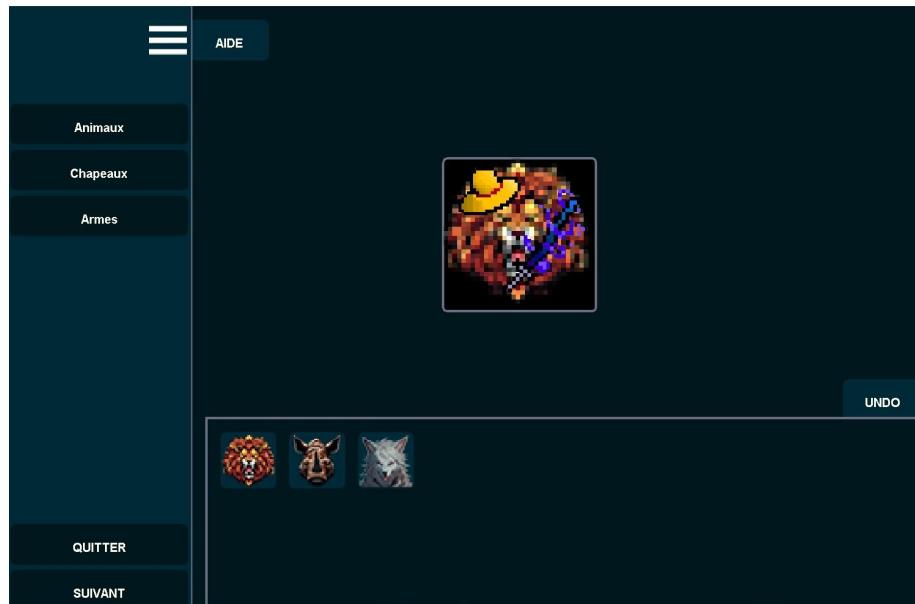
C'est à partir de cette interface que le joueur va pouvoir naviguer dans les différentes sections de l'extension. Ainsi, on commence naturellement par choisir l'animal et pour cela, le joueur clique sur les trois bandes présentes en haut à gauche :



Menu de sélection des animaux, une fois que le joueur a cliqué sur les 3 bandes, puis sur la section « Animaux »

Les sections « Animaux », « Chapeaux » et « Armes » jouent sur le visuel.

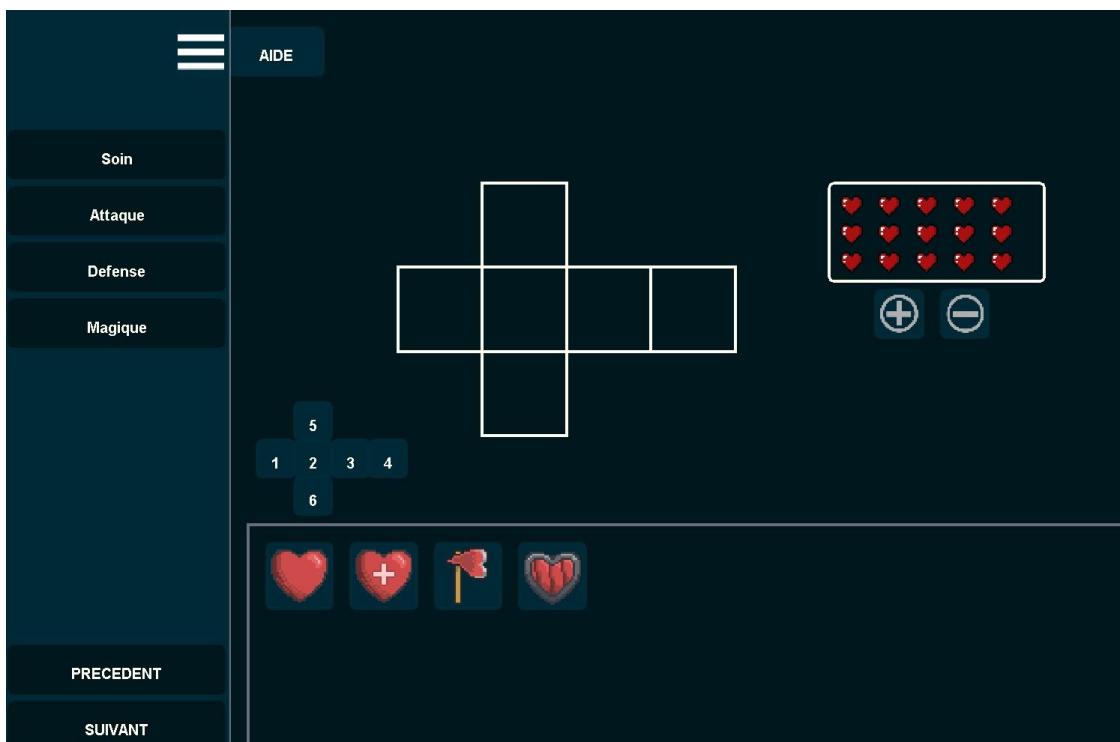
On notera la possibilité pour le joueur de placer le chapeau et l'arme où il le désire à l'intérieur de la case grâce aux flèches directionnelles.



Exemple de héros personnalisé après avoir choisi un chapeau et une arme

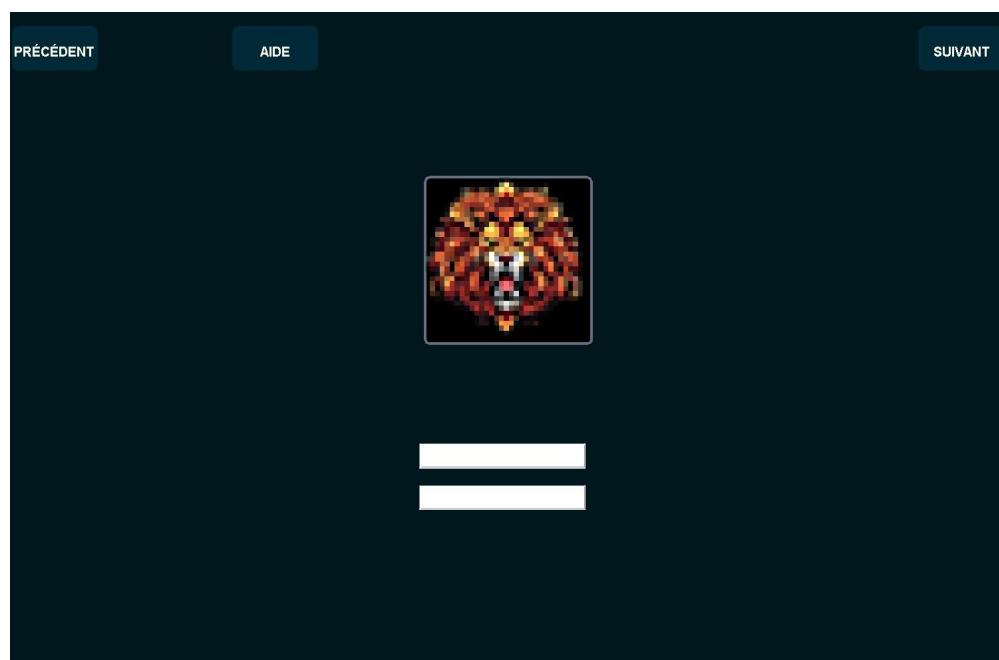
Pour passer à la section suivante, le joueur peut cliquer sur « SUIVANT » en bas à gauche du menu, ce qui le redirigera vers la page de sélection des capacités.

C'est à partir de cette interface que le joueur peut désigner à chaque case du dé de son héros personnalisé une capacité de son choix. Cependant, pour éviter que le joueur n'abuse des meilleures capacités, nous avons décidé de mettre un système de points de compétences. Le joueur démarre avec 20 points de compétences. Ces derniers seront décrémentés d'une certaine somme en fonction du niveau de la capacité que le joueur attribue à son héros Custom, mais également, ces points de compétences seront décrémentés si le joueur décide d'ajouter des points de vie à son héros. Le joueur a accès aux différents types de capacité sur le menu vertical à sa gauche.



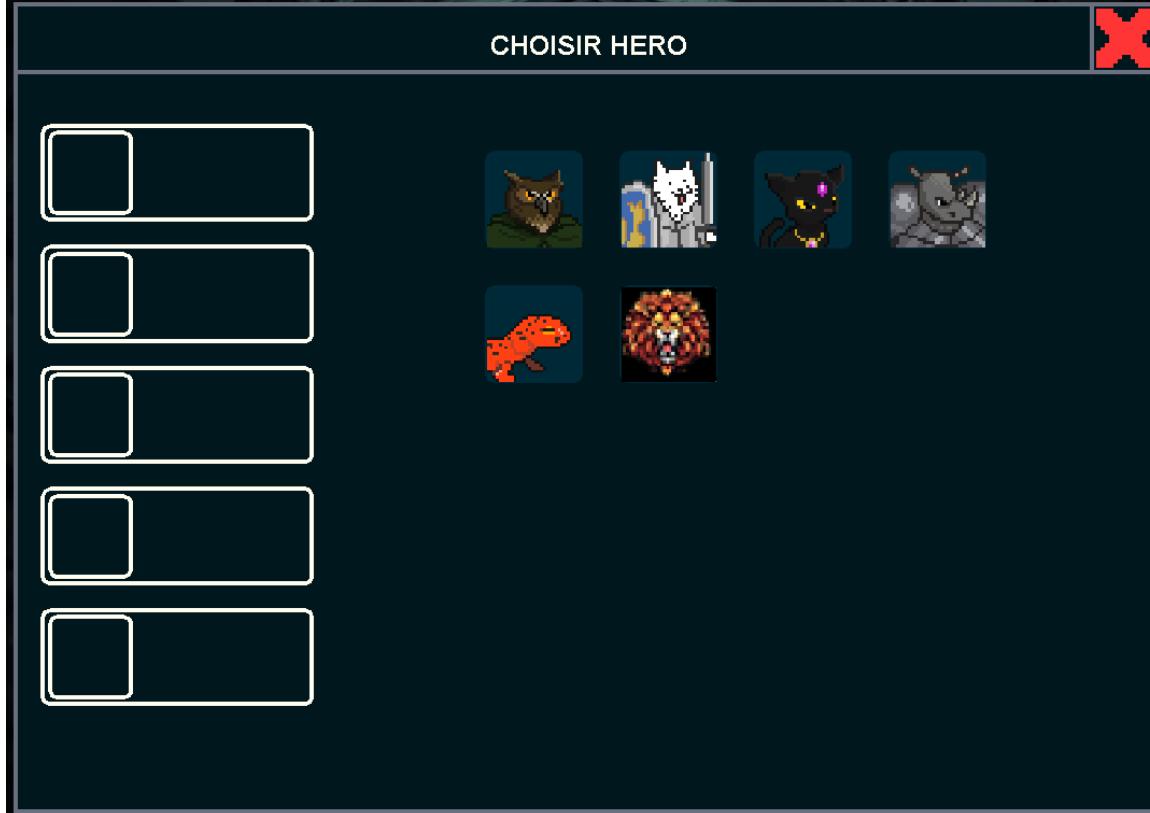
Menu Custom, section Choix des capacités

Une fois l'attribution au héros de son visuel et de ses capacités, le joueur a la possibilité de donner un nom à son héros, ainsi qu'une classe. Noter que la classe donnée par le joueur n'aura aucun impact par la suite.



Menu Custom, section Nom et Classe. Le premier champ permet l'attribution du nom, le deuxième celui de la classe

Une fois le héros sauvegardé, il pourra être choisi pour faire partie de la formation à la place d'un autre héros.



Interface de sélection des héros; le héros custom est bien sélectionnable

Partie technique :

L'extension Custom nous a demandé beaucoup de travail, notamment car il fallait trouver les moyens d'appliquer la sauvegarde du héros tant sur le plan visuel que sur le plan des capacités. Par exemple, le visuel d'un héros de base est un sprite que nous avons créer à la main puis enregistré sous format .png. Cependant, le fait que le joueur puisse disposer des éléments comme il le souhaite à l'intérieur de la case contenant le sprite du héros Custom nous constraint naturellement à ne pouvoir enregistrer le visuel qu'une fois que ce dernier ne changera plus. Pour cela, nous avons décidé d'utiliser la fonction tabPixels, une fonction native de swing qui permet d'associer une « capture d'écran » à une chaîne de caractères.

Voir l'exemple ci-dessous :

```
else if(buttonFinish.getBounds().contains(x,y)){ // action lorsque l'on appuie sur le bouton final
    heroPanel.tabPixels(heroPanel.getX(), heroPanel.getY(), name); //enregistre l'image
    deCustom = dicePage.getDeCustom(); //recupère le dé
    herosCree = createHeroCustom(pdv, deCustom, new Color(r: 0, g: 23, b: 30), namepage.getName(), namepage.getClasse(), heroPanel.getPath());
    EtatsJeu.setEtatJeu(MENU);
    /* if (herosCree){
        EtatsJeu.setEtatJeu(MENU);
    }*/
}
```

Partie de la fonction mouseClicked de la classe CustomFinalPage; chaque ligne permet la sauvegarde d'une partie de l'entité du héros personnalisé qui vient d'être créée

Ainsi, la chaîne de caractères utilisée est le nom donné au héros par le joueur. Le dé, qui avait été stocké dans la classe CustomDicePage est ici attribué en argument de la fonction creationHeroCustom qui finalise la création de l'entité. Enfin, EtatsJeu.setEtatJeu(MENU) permet le retour au menu principal. D'ailleurs, c'est aussi de cette manière que le joueur est permis de naviguer entre les différentes pages du menu de l'extension Custom, mais aussi des menus en général.

Développement du projet :

Cette partie traite plutôt de l'aspect organisation, développement du projet. Il y aura donc moins de mention de technicité au profit du ressenti du groupe. De plus, quelques détails utiles sont fournis dans les comptes rendus de chaque semaine, qu'il est conseillé de lire en même temps que cette partie pour une meilleure compréhension (ces derniers se trouvent dans le dossier Documentation du projet, le même dossier d'où vient ce rapport).

Début du projet :

Le début du projet a directement été relativement compliqué. En effet, si la première semaine a été plutôt tranquille, chaque membre ayant réussi à remplir la tâche qui lui avait été assignée, on s'est vite rendus compte que nous allions probablement manquer de temps, notamment car 4 membres sur 5 sont issus de la double licence Biologie/Informatique. Malgré tout, le début du projet nous a surtout permis de comprendre l'importance de l'organisation lorsque l'on possède un temps très limité. Pour mieux définir nos objectifs pour la semaine suivante, nous avons également décidé de partager sur les comptes rendus hebdomadaires le ressenti global du groupe, notamment sur l'avancée. Ces ressentis étaient bien sûr résumés par la personne en charge de la rédaction du compte rendu, à partir des ressentis personnels partagés sur le groupe WhatsApp.

Enfin, il a été décidé que la répartition des tâches pour chaque semaine soit faite par la personne rédigeant les comptes rendus hebdomadaire, cette dernière prenant bien sûr en compte toutes les demandes de chaque groupe, afin de maintenir une cohérence et éviter le plus de chevauchements possibles.

Mais si le début du projet, malgré quelques problèmes techniques, est synonyme de bonne avancée, il s'avère que nous nous sommes très vite heurtés à un autre gros problème de taille par la suite : **l'importance de la communication**.

Milieu de projet :

Si la majorité de cette période (Semaine 4 à Semaine 7) a été marquée par une avancée globale satisfaisante, c'est véritablement lors de la huitième semaine que le développement de notre projet va prendre un tournant majeur. En effet, un des objectifs de cette semaine 8 était de pouvoir supprimer les duplicata de fonctions qui font exactement la même chose. Cependant, durant ce premier nettoyage, nous avons remarqué que deux classes avaient exactement le même intérêt, des

fonctions similaires, avec des objectifs similaires. Or, les deux classes étaient utilisées pour faire fonctionner le jeu, signifiant ainsi que **supprimer l'une d'entre elles casserai le jeu.** Ces deux classes étant **Jeu.java et Combat.java.** C'est à partir de ce moment que l'on comprit que les membres du groupe ayant travaillé sur ces deux classes auraient du mieux communiquer sur les objectifs des classes et des fonctions qu'ils programmaient, ce qui aurait évité la multiplication d'utilisations de deux classes portant la même fonction. Cela a malheureusement donné lieu à quelques querelles, qui ont été résolues par un appel Discord dans lequel une solution a été trouvée, satisfaisant les deux partis. **Cette solution a été de transférer toutes les fonctions de Combat dans Jeu, et de simplement supprimer Combat par la suite.** Cela a impliqué un énorme travail de refonte du code, et a naturellement retardé grandement l'avancée du projet.

Cette expérience peut être vue comme bonne ou mauvaise selon la perspective de chacun, notre groupe en retient surtout l'importance de communiquer, de mettre son égo de côté au profit du succès du groupe, et de ne pas abandonner à la première erreur qui survient.

C'est donc avec cette mentalité là que nous entamions la fin du projet.

Fin de projet :

Fort de nos nouveaux acquis, le projet a pu se dérouler tant bien que mal comme il était prévu de se dérouler. Notre double licence ne nous a pas laissé beaucoup de temps, puisque c'est durant cette période que nous avons enchainé plusieurs partiels et autres comptes rendus de Travaux Pratiques en laboratoire. La plupart des extensions prévues ont été implémentés (voir section Annexe.Résumé des extensions) et nous avions enfin une vision finie de ce à quoi ressemblerai notre projet. C'est également au début de cette période qu'ont commencé les phases de déboggage, de réorganisation du code pour ressembler au mieux au modèle MVC. De plus, c'est aussi à partir de cette période que les membres du groupe ont pu commencé à travailler directement ensemble sur une partie du projet, comme par exemple le système de combat ou encore l'extension Custom, ce qui a encore plus facilité la communication et donc le bon déroulement de ces développements.

Un document Word partagé a été crée dans lequel les bugs trouvés sont répertoriés et marqués comme réglé ou non, permettant à tout le monde de voir s'il peut régler un problème rapidement.

À l'heure de rédaction de ce compte rendu, encore quelques détails continuent d'être travaillés.

Dernier mot :

Ce projet nous a certainement permis à chacun de devenir un peu plus expérimentés en terme de programmation, mais surtout en termes d'organisation, et de travail de groupe. Les expériences vécues durant le développement de ce projet montre concrètement qu'il peut y avoir des problèmes autres que ceux montrés sur le terminal, et que l'important est de communiquer avec ses collègues pour éviter la frustration, et avancer le plus loin possible. La consigne pour notre groupe était de programmer un jeu du style Slice & Dice, une consigne finalement, qui peut laisser place à beaucoup d'imagination et de créativité. Mais l'imagination ne possède pas les limites que nous, en tant qu'étudiants en double licence, possédons. Il faut alors savoir faire la part des choses, faire les sacrifices nécessaires qui, sur le long terme, pourront permettre au jeu de briller sur ce qu'il propose, plutôt que de bâcler ses fonctionnalités. Il est évident que nous aurions voulu aller plus loin, mais nous sommes globalement satisfaits de jusqu'où nous avons réussi à amener ce projet avec le peu de temps qui nous était alloué.

Nous remercions l'équipe pédagogique de nous avoir permis de profiter de cette expérience malgré les hauts et les bas. Et nous remercions chacun des membres du groupe qui n'ont pas abandonné, malgré toutes difficultés rencontrées, qu'elles aient été écrites en rouges sur le terminal, ou qu'elles aient été intérieures de leurs personnes même.

Annexe - Résumé des extensions :

Implémentées :

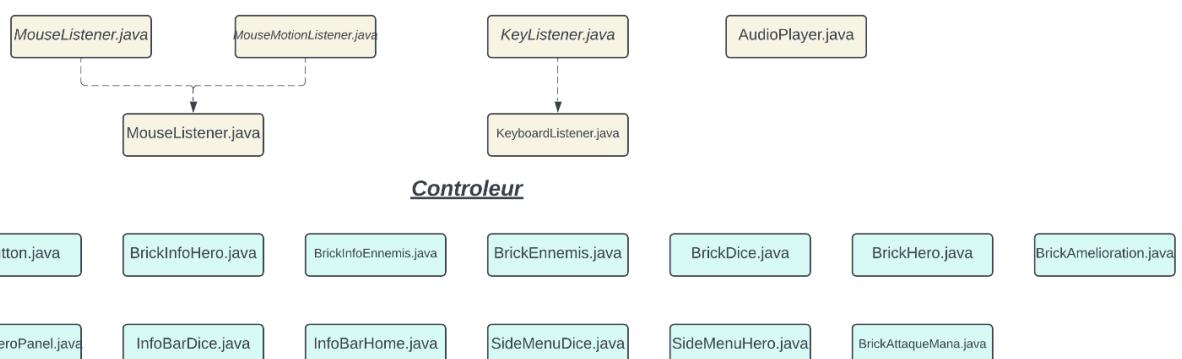
- Sauvegarde
- Custom : héros personnalisable
- Améliorations
- Nouvelles Capacités
- Boutique de Capacités
- Boutique d'équipements
- Changement d'arrière-plans qui suit la progression dans la partie
- Évènements aléatoires
- Poison
- Succès
- Cinématique de début

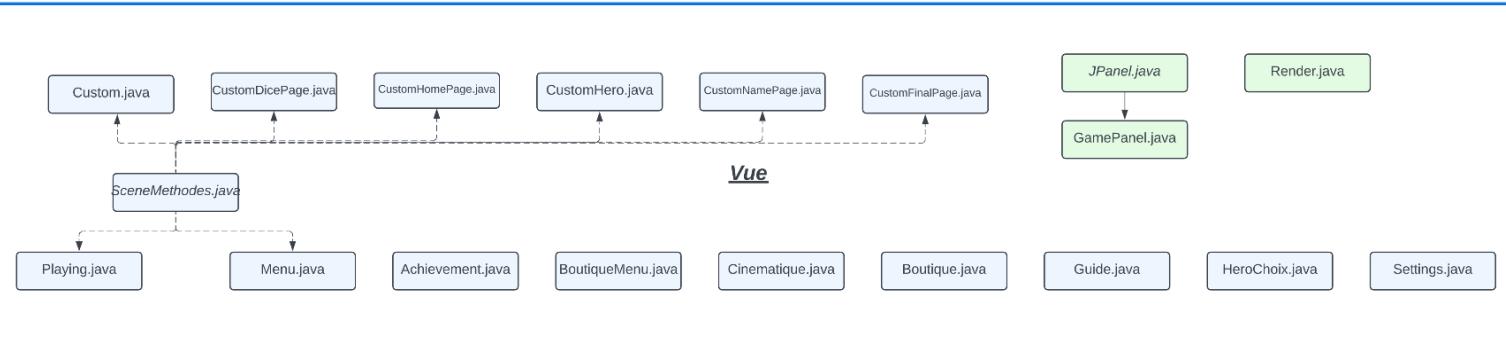
Non implémentés :

- QTE : des séquences de commandes que le joueur doit effectuer au clavier, à la souris, pour permettre à ses héros d'infliger plus de dégâts, ou encore d'accumuler plus de dégâts par exemple.
- Affichage des stats à la fin de la partie
- Vidéo tutoriel qui montre comment jouer au jeu et accessible pendant la partie
- Effets sonores et animations pour l'utilisation des capacités
- Cinématique de fin

Plus que le manque de compétences ou les problèmes techniques, c'est surtout par manque de temps que nous n'avons pas pu implémenter ces extensions.

Annexe - Diagramme des classes :





Seuls les diagrammes du Contrôleur et de la Vue peuvent être montrés ici, le diagramme des classes du Modèle étant beaucoup trop grand pour être observé agréablement, même découpé.