Jalon 2 : Conception Préliminaire

Architecture du programme :

Organisation :

Dans l'organisation du projet, nous avons choisi de mettre en place une structure conventionnelle avec le dossier principal 'src/game' qui regroupe l'ensemble du code source du jeu. À la suite, on trouve plusieurs sous-dossiers, notamment 'models' qui héberge les modèles du jeu tels que Fighter et Dragon. Il y a également un dossier 'views' dédié à l'interface du projet, 'controllers' qui gère la logique et le comportement, ainsi qu'un dossier appelé 'utils' contenant d'autres fichiers de code qui ne sont pas nécessairement liés au jeu, mais demeurent utiles pour le projet. Enfin, le dernier dossier est 'resources' qui stocke les images et les éléments graphiques du jeu au format .jpg et .png.

Project Organization

Project Structure

- **src**: Contains all the source code for the Java game.
  - **game**: Main game code.
    - **models**: Classes representing different objects in the game (Dragons & Fighters).
    - **views**: Different views or screens of the game (Labyrinth, RightSidePanel, FireAndShield, etc.).
    - **controllers**: Classes handling user input and managing game logic (DragonActionAI, CharactersMovesManage, Sound, etc.).
    - **utils**: Utility classes or helper functions used throughout the game (PicturePaths, MapMaths).
    - **resources**: External resources needed for the game, such as images (Packages for each Dragon/Fighter).

- ## Controllers :

Il s'agit du répertoire chargé de toutes les actions, de la logique principale, ainsi que du comportement du projet, avec son propre package :

```
package src.mygame.Controllers;
```

À l'intérieur de ce dossier, on trouve quatre fichiers .java.

1. CharactersMovesManage.java :

Ce fichier est responsable de tous les mouvements des deux modèles, "fighter" et "dragon". Il contient des méthodes qui gèrent les mouvements (les sprites d'image) ainsi que leurs

déplacements. De plus, il abrite la méthode responsable de la collision du combattant avec les murs du labyrinthe. Il comprend également des méthodes spécifiques aux mouvements dans les combats.

```
public void moveFighter(int Choice, int dx, int dy , labyrinth labyrinthImportant)
public void ShowFighter(String nomImage, String nomImage2, labyrinth labyrinthInstance)
public void moveAttack( int Choice, labyrinth labyrinthInstance)
```

Ce fichier fait appel à deux classes, PicturesPath, qui contient les chemins de toutes nos images, et utilise également des classes du dossier Views pour visualiser les mouvements et gérer la collision.

```
import src.mygame.Views.*;
import src.mygame.Utils.PicturesPath;
```

2. **DragonActionIA.java :**

Ce fichier représente l'intelligence artificielle appliquée au dragon pour ses propres mouvements dans le combat. Cette intelligence est réglée à travers une boucle de switch, des conditions if, des boucles while et for, ainsi que quelques méthodes de randomisation codées dans le fichier MapMaths.java du dossier Utils.

```
public void DragonActionAI(labyrinth labyrinth)
```

Ce fichier fait appel à WelcomingPage.java pour connaître le niveau de la partie, utilise également les fichiers Fighter et Dragon pour appliquer l'intelligence artificielle, ainsi que MapMaths pour les méthodes de randomisation utilisées dans le fichier.

```
import src.mygame.WelcomingPage;
import src.mygame.Models.TheDragon.Dragon;
import src.mygame.Models.TheFighter.FighterCaracter;
import src.mygame.Views.labyrinth;
import src.mygame.Utils.MapMaths;
```

3. **GameTimers.java :**

Ce fichier est le cœur centrale de l'organisation, de la chronologie , du dynamisme de jeux et des taches dans le jeu. Il est responsable de tout les délais et de la gestion de la logique du jeu, et va les variables 'timer' .

Ce fichier fait appel à plusieurs ressources dans le projet car il pose la main sur tout les mécanismes temporaire dans le projet

```
import src.mygame.Views.RightSidePanel;
import src.mygame.Views.FireAndShieldManage;
import src.mygame.Views.labyrinth;
import src.mygame.Models.TheDragon.Dragon;
import src.mygame.Models.TheFighter.FighterCaracter;
import  src.mygame.Views.winOrLose;
```

4. Sound.java

Ce fichier est responsable des instruments sonores dans notre projet, générant tous les sons dans le jeu.

```
public Sound() {
 public void setSoundFile(int i) {
 public void playSound(int i) {
 public void stopSound() {
 public long getClipDuration() {
```

En résumé, ces fichiers jouent des rôles cruciaux dans la gestion des mouvements des personnages, de l'intelligence artificielle du dragon, la gestion temporelle du jeu et de la gestion des sons.

• Models :

À l'intérieur de ce répertoire, deux autres dossiers sont présents : "The Fighter" et "The Dragon". Ces dossiers jouent le rôle d'hébergeurs pour les classes du combattant (fighter) et du dragon, abritant l'ensemble de leurs attributs et méthodes spécifiques. Ces subdivisions permettent une organisation claire et distincte des fonctionnalités associées au combattant et au dragon dans le contexte global du projet. , avec ses propres packages :

```
package src.mygame.Models.TheFighter;
package src.mygame.Models.TheDragon;
```

Les fichiers FighterCaracter.java et Dragon.java ont été codés en utilisant le design pattern Singleton. Cette approche a été adoptée dans le but d'éviter la création de multiples instances, assurant ainsi qu'il n'y a qu'une seule instance de chaque objet, que ce soit pour le combattant (fighter) ou le dragon. Ce choix de conception garantit une gestion efficace et unifiée de ces entités au sein du système, évitant toute duplication non nécessaire et assurant une cohérence dans la manipulation de ces classes tout au long du projet.

```
public class FighterCaracter {
public class Dragon {
```

Ces deux fichiers font appel aux classes suivantes :

```
import src.mygame.Models.TheDragon.Dragon;  // pour effectuer les attaque et les défense
import src.mygame.Views.labyrinth;   // pour visualiser dans labyrinthe
import src.mygame.Views.winOrLose;   // dans la victoire ou la perte de joueur
import src.mygame.Controllers.Sound; // pour faire du son lors d'un mouvement du combattant
```

- ## Utils

À l'intérieur de ce répertoire, on découvre des fichiers qui, bien que ne jouant pas un rôle central dans le projet, demeurent néanmoins utiles. Par exemple, le fichier MapMaths.java est dédié aux opérations mathématiques dans les matrices, utilisées pour la création du labyrinthe, et contient également des méthodes utiles pour le fichier DragonIA.java. Un autre fichier présent est PicturePaths.java, regroupant un ensemble de chemins (paths) pour les ressources du jeu. Ces fichiers, bien que périphériques, contribuent de manière significative à la fonctionnalité globale du projet en fournissant des utilitaires mathématiques et des références structurées aux ressources nécessaires. Son propre package est :

```
package src.mygame.Utils;
```

les méthodes présentes :

```
public int[][]  loadMap( String Level){
public int[][] CreatTheBigMatrix(int Littlematrix[][]){
public int[][] transposeMatrix(int[][] matrix) {
public int generateRandomZeroOne(float V) {
```

- ## Views :

Il s'agit du répertoire chargé de tout le volet interface graphique dans notre projet , avec son propre package :

```
package src.mygame.Views;
```

1. Bonus.java :

Ce fichier est associé à la composante graphique des bonus du combattant, notamment la présence d'étoiles qui représentent des bonus de vie. Par exemple, si le combattant passe à proximité d'une étoile, sa barre de vie augmente de 20 points.

```
public void Bonus(labyrinth labyrinth) {
```

Ce fichier interagit exclusivement avec labyrinth.java pour dessiner les étoiles de bonus, les plaçant stratégiquement dans des emplacements vides du labyrinthe.

2. FireAndShieldManage.java :

Ce fichier assume la responsabilité de la composante graphique de tous les boucliers de défense et des feux d'attaques associés à chaque niveau du jeu. Il gère l'affichage visuel de ces éléments, assurant une représentation graphique cohérente et adaptée à la progression du joueur dans les différents niveaux du jeu.

Ce fichier a été élaboré en utilisant un modèle de conception Factory, dans le but de maintenir l'isolation de nos ressources (variables de chaîne contenant les chemins de nos images) vis-à-vis de l'utilisateur. En employant ce modèle, nous avons créé une structure qui permet à l'utilisateur d'accéder aux ressources nécessaires sans la nécessité de manipuler directement les détails complexes des chemins d'accès. Cette approche favorise une abstraction efficace et simplifie l'utilisation des ressources graphiques dans l'ensemble du projet.

Dans ce fichier, le concept d'« Implements » a également été appliqué pour étendre une classe, similaire à ce que réalise « extend », mais cette fois-ci pour une interface. L'interface mère est définie comme suit :

```
interface CharacterFactory {
   String getFireImage();
   String getShieldImage();
   String getInversedFireImage();
   String getInversedShieldImage();
}
```

Chaque classe représentant différents niveaux du dragon a ensuite été implémentée comme suit :

```
class EasyLevelDragonFactory implements CharacterFactory {
class MediumLevelDragonFactory implements CharacterFactory {
class HardLevelDragonFactory implements CharacterFactory {
```

Chacune de ces classes s'engage à implémenter les méthodes spécifiées dans l'interface CharacterFactory. Ainsi, cette approche utilise le mécanisme d'implémentation d'interface pour garantir que chaque niveau du dragon fournit une mise en œuvre concrète cohérente des fonctionnalités requises par l'interface. Cette utilisation judicieuse d'interfaces et d'implémentations garantit une structure modulaire et extensible dans le code.

3. Labyrinth.java :

Ce fichier constitue le noyau central de notre projet, orchestrant l'utilisation de l'ensemble des fichiers de code. Il se charge de tracer le labyrinthe, d'initialiser le jeu en affichant les combattants, et de gérer les événements clés tout en mettant en œuvre le système sonore. En somme, il joue un rôle fondamental en intégrant toutes les fonctionnalités essentielles du projet et en assurant la cohérence de son fonctionnement global.

Étant donné que ce fichier requiert l'exécution de plusieurs méthodes pour amorcer le jeu, il effectue automatiquement des appels à divers fichiers :

```
import src.mygame.Utils.*;
import src.mygame.WelcomingPage;
import src.mygame.Controllers.*;
import src.mygame.Models.TheDragon.Dragon;
import src.mygame.Models.TheFighter.FighterCaracter;
```

Ces importations englobent un éventail de fichiers, couvrant des utilitaires, la page d'accueil, les contrôleurs, ainsi que les modèles du dragon et du combattant. Par cette inclusion, le fichier principal établit un lien essentiel avec ces composants, assurant une coordination harmonieuse de l'ensemble des fonctionnalités nécessaires pour lancer le jeu.

4. RightSidePanel.java :

Le fichier RightSidePanel constitue une section spécifique de la fenêtre de jeu, plus précisément la partie droite. Il englobe toutes les informations relatives à la partie en cours, comprenant des détails tels que le niveau, les images des combattants, leurs noms, et leurs points de vie. De plus, il intègre des données dynamiques telles qu'un chronomètre indiquant le temps investi dans le combat, ainsi que les points de vie des combattants qui se mettent à jour à chaque bonus ou dommage subi. Ce panneau latéral droit offre ainsi une visualisation complète et en temps réel de divers éléments cruciaux pour une expérience de jeu immersive.

Cette classe utilise le concept d'héritage en héritant d'une classe mère prédéfinie par Java.

```
public class RightSidePanel extends JPanel
```

La classe RightSidePanel hérite des fonctionnalités de la classe JPanel, une classe prédéfinie dans Java pour la création de panneaux graphiques. En utilisant l'héritage, RightSidePanel peut bénéficier des attributs et méthodes de JPanel, tout en ayant la possibilité de définir ses propres fonctionnalités spécifiques à la représentation graphique de la partie droite dans le contexte du jeu.

Comme ce fichier à besoin des propriétés des combattants donc il fait appel aux deux classes TheCaractere.java et Dragon.java, aussi fait appel au GameTimers.java afin d'implémenter le chronomètre.

```
import src.mygame.Models.TheDragon.Dragon;
import src.mygame.Models.TheFighter.FighterCaracter;
import src.mygame.Controllers.*;
```

5. WinOrLose.java :

Ce fichier détient la responsabilité centrale de l'interface graphique qui se manifeste suite à la victoire ou à la défaite du personnage. Il gère l'affichage visuel des éléments spécifiques à ces situations, offrant une expérience utilisateur immersive et réactive. En réaction aux événements du jeu, cette interface graphique peut présenter des messages, des animations ou d'autres éléments interactifs, contribuant ainsi à la rétroaction visuelle et à l'engagement du joueur en fonction du dénouement de la partie.

• WelcomingPage.java :

Le fichier welcomingPage.java marque le point de départ de l'exécution du projet. Il agit comme une interface graphique initiale, se manifestant comme la première interaction avec l'utilisateur.

Lors de la compilation, cette interface propose des boutons permettant à l'utilisateur de sélectionner le niveau de difficulté ainsi que le combattant. En fonction de ces choix, le programme génère un labyrinthe spécifique, ajuste les propriétés des autres fichiers, et déclenche les mécanismes nécessaires pour débuter le jeu conformément aux préférences définies par l'utilisateur. En résumé, welcomingPage.java orchestre les paramètres initiaux du projet, offrant une interface conviviale pour personnaliser l'expérience de jeu dès le départ.

Son propre package est :

```
package src.mygame;
```

• Makefile :

Dans cette section du projet, le fichier makefile simplifie le processus de compilation en référençant l'ordre chronologique des fichiers, évitant ainsi la nécessité d'une compilation classique avec les commandes `java` et `javac`. De plus, il gère également la suppression des fichiers `.class` lors de la fermeture du jeu, contribuant ainsi à une gestion efficace des ressources du projet. Cette approche automatisée facilite le développement en assurant une séquence de compilation fluide et en simplifiant la maintenance du code source.

- ## Diagramme des séquences :

Page d'Accueil :

1) L'utilisateur interagit avec l'interface graphique pour choisir un personnage et un niveau de difficulté.
2) La classe WelcomingPage (ou équivalente) réagit à ces sélections.

Initialisation du Jeu :

1) La classe WelcomingPage crée une instance de FighterCaracter en utilisant le pattern Singleton et récupère également le choix du niveau de difficulté.
2) Elle crée une instance de la classe EasyLevelDragonFactory, MediumLevelDragonFactory, ou HardLevelDragonFactory en fonction du niveau de difficulté choisi, utilisant ainsi le pattern Factory.

Création du Labyrinthe et des Personnages :

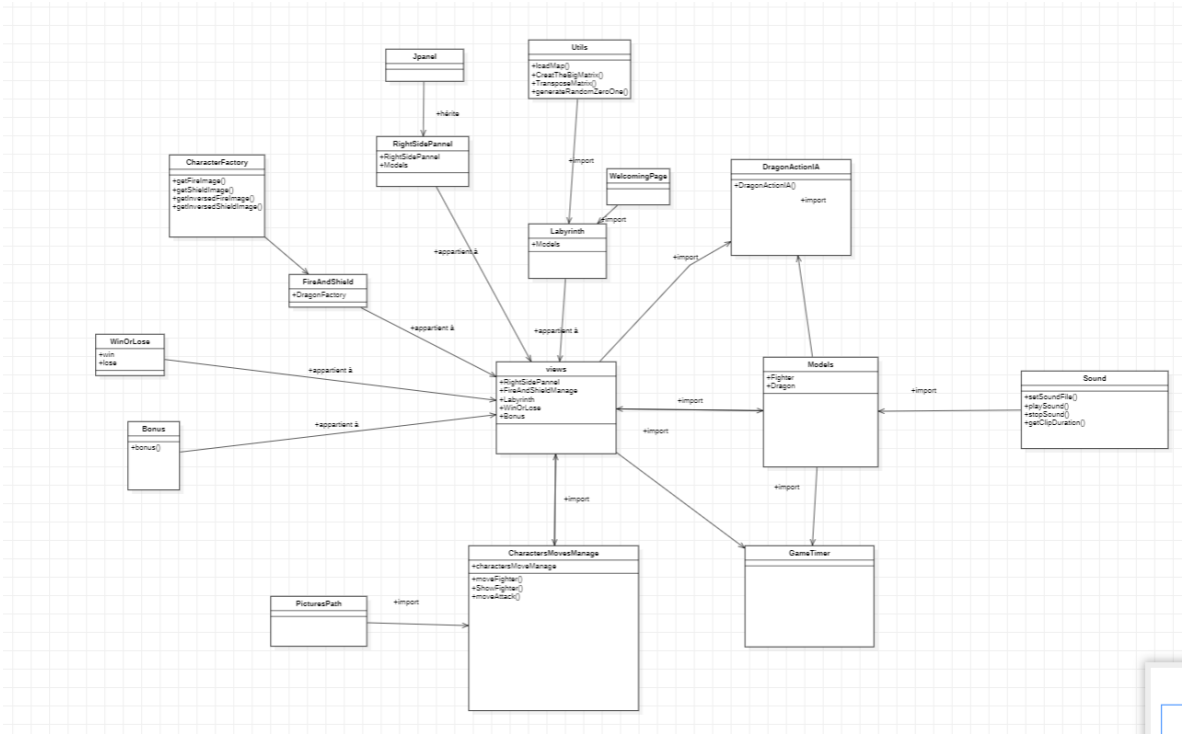
1) La classe Labyrinth crée un labyrinthe pour le niveau choisi.
2) La classe FighterCaracter (instance unique grâce au Singleton) est positionnée dans le labyrinthe.
3) La classe Dragon (instance unique grâce au Singleton) est également positionnée.

Affichage Initial :

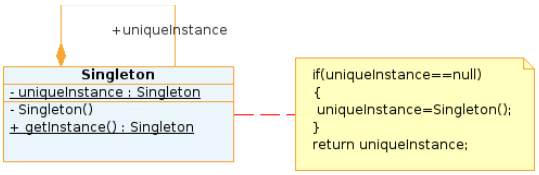
1) La classe Labyrinth informe la classe RightSidePanel pour mettre à jour les informations sur le panneau droit, affichant les détails du combattant, du dragon, et d'autres informations du jeu.
2) Après l'initialisation du jeu, la classe GameTimers est activée pour démarrer les compteurs et surveiller le temps de jeu.
3) Lorsqu'une condition de victoire ou de défaite est détectée (par exemple, le joueur atteint la sortie du labyrinthe ou perd toutes ses vies).
4) La classe WinOrLose réagit à cet évènement en affichant l'interface graphique appropriée, indiquant si le joueur a gagné ou perdu.
5) Si le joueur a gagné, WinOrLose affiche un message de victoire et proposer des options pour rejouer ou quitter le jeu.

- ## Diagramme de Classes :

Le diagramme UML du projet est joint en annexe. De par la difficulté d'organiser (et de lire) un diagramme complet du projet et de toutes ses classes avec tous leurs liens, le diagramme fourni est une version simplifiée.

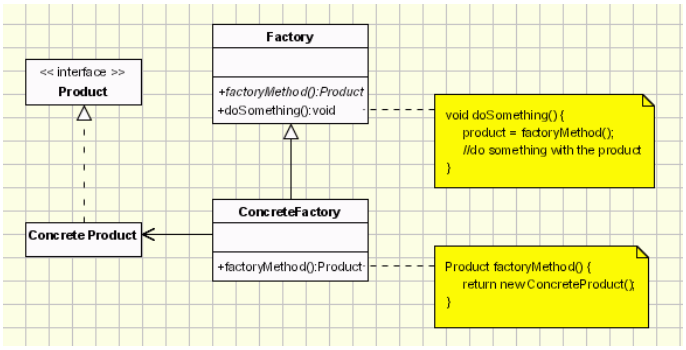


1. **Singleton :**
    - Le pattern Singleton est utilisé de manière significative dans le package "models" pour les classes FighterCaracter et Dragon. Ces classes représentent respectivement le combattant et le dragon du jeu. L'implémentation du Singleton garantit qu'il n'y a qu'une seule instance de chaque modèle dans tout le programme. La méthode **getInstance()** est utilisée pour accéder à cette instance unique. Cela assure une gestion centralisée et cohérente des données liées au combattant et au dragon, évitant ainsi toute duplication non nécessaire.



2. **Factory**
    - Le pattern Factory est implémenté dans la classe FireAndShieldManage du package "views". Cette classe gère la composante graphique des boucliers de défense et des

feux d'attaques associés à chaque niveau du jeu. Trois classes, EasyLevelDragonFactory, MediumLevelDragonFactory, et HardLevelDragonFactory, sont utilisées pour fournir différentes implémentations des images des boucliers et des feux d'attaques en fonction du niveau du dragon. L'utilisation de ce pattern offre une séparation claire entre la création des instances et leur utilisation, facilitant la maintenance et l'évolutivité du code.



- **Types primitifs**

Dans notre projet, comme il est codé en java donc les types pritmitifs comme int32 et int64 ne sont pas des types primitifs en Java. En Java, les types primitifs qu'on a utilisés sont int, long, etc.

int : Il est utilisé dans les deux classes Fighter et Dragon afin de définir tout les propriétés des combattants : Damage, Defense, Xfighter, Yfighter, Death, Score …

String : Il est utilisé dans les deux classes Fighter et Dragon aussi afin de définir les nomes des combattants, aussi dans le fichier welcomingPage pour définir le niveau (Thelvel & TheCharactere)

Char : il est utilisé dans le fichier labyrinth.java et CharactersMovesManage.java afin d'indiquer la direction des deux combattants et gérer leurs mouvements.

```
public char SideDragon='L', SideFighter='R';
```

Long : l'utilisation de la variable long est utilisé dans le fichier sound.java dans la dernière méthode afin de retourner la longueur de le clip sonneur

```
public long getClipDuration() {
```

```
        if (clip != null) {
            return clip.getMicrosecondLength() / 1000;
        }
        return 0;
    }
}
```

- **Tableaux**

Les tableaux EasyMatrix, MediumMatrix et HardMatrix sont des exemples clés de l'utilisation des tableaux en Java pour organiser des données complexes. Ces tableaux stockent les configurations des labyrinthes pour différents niveaux de difficulté du jeu, démontrant comment les tableaux multidimensionnels peuvent être utilisés pour représenter des structures de données élaborées. Cela souligne l'efficacité des tableaux pour gérer des ensembles de données organisés, permettant une manipulation et un accès efficaces aux informations nécessaires pour la logique du jeu.

- **L'héritage**:

L'héritage est un principe de la programmation orientée objet qui permet à une classe d'hériter des caractéristiques (propriétés et méthodes) d'une autre classe. Cette approche favorise la réutilisation du code et la création d'une hiérarchie de classes. L'héritage est mis en œuvre, par exemple, dans RightSidePanel qui hérite de JPanel, profitant ainsi des fonctionnalités générales de gestion d'interface graphique. L'héritage permet d'utiliser les méthodes et attributs de JPanel sans les redéfinir. La classe étend ses fonctionnalités avec des membres spécifiques tels que labyrinthRightSidePanel, gameTimers, etc., adaptés aux besoins de l'application. Cela favorise la réutilisation du code et la modularité.

Le concept d'« Implements » a également été appliqué pour étendre une classe,
similaire à ce que réalise « extend », mais cette fois-ci pour une interface. L'interface mère est définie comme suit :

```
 interface CharacterFactory {
    String getFireImage();
    String getShieldImage();
    String getInversedFireImage();
    String getInversedShieldImage();
}
```

Chaque classe représentant différents niveaux du dragon a ensuite été implémentée comme suit :

```
class EasyLevelDragonFactory implements CharacterFactory {
```

```
class MediumLevelDragonFactory implements CharacterFactory {
class HardLevelDragonFactory implements CharacterFactory {
```

Chacune de ces classes s'engage à implémenter les méthodes spécifiées dans l'interface CharacterFactory. Ainsi, cette approche utilise le mécanisme d'implémentation d'interface pour garantir que chaque niveau du dragon fournit une mise en œuvre concrète cohérente des fonctionnalités requises par l'interface. Cette utilisation judicieuse d'interfaces et d'implémentations garantit une structure modulaire et extensible dans le code

- **La gestion des dépendances**

L'objectif est de minimiser les liens entre les différentes classes et objets. Cette approche est illustrée par la manière dont les classes interagissent de façon ciblée et efficiente. Par exemple, la classe CharactersMovesManage interagit étroitement avec labyrinthe pour gérer les mouvements des personnages, mais sans dépendre excessivement d'autres composants. Cette structuration permet de garder chaque classe focalisée sur sa propre responsabilité, facilitant ainsi la maintenance et l'évolution du code. Une gestion soignée des dépendances assure une meilleure modularité et une plus grande flexibilité pour les futures modifications.

- **Exception:**

L'utilisation d'exceptions dans la méthode setSoundFile(int i) est cruciale pour anticiper et gérer d'éventuelles erreurs pendant la lecture d'un fichier audio. Si le chemin du fichier est nul ou si des problèmes surviennent lors de la lecture, une exception est déclenchée. Le bloc catch capture cette exception, affiche un message d'erreur, et imprime une trace détaillée pour faciliter le débogage. Cette approche renforce la robustesse du code, facilite la gestion des erreurs, et contribue à assurer une expérience utilisateur fiable.

- **Chaine de caractères :**

L'utilisation de chaînes de caractères est prédominante pour représenter des informations textuelles telles que les chemins d'images, les niveaux de jeu, les personnages sélectionnés, etc.

Dans l'interface CharacterFactory, les méthodes définissent des chemins d'images via des chaînes pour différentes actions des personnages.

La classe WelcomingPage utilise des chaînes de caractères pour stocker les informations sur le niveau (theLevel) et le personnage (theCharacter). Les méthodes getTheLevel() et getTheCharacter() renvoient ces chaînes respectivement, facilitant l'accès à ces informations.

La classe PicturesPath définit des membres de classe en tant que chaînes représentant les chemins d'accès aux images de différents personnages. Les tableaux de chaînes, comme

imagePathsPirateRun, stockent les chemins d'images associés à des actions spécifiques du personnage.

Dans la classe CharactersMovesManage, des tableaux de chaînes tels que picturePaths, picturePathsAttack, etc., stockent les chemins d'images liés aux mouvements et aux attaques des personnages. Les variables theLevel et theCharacter sont également des chaînes de caractères représentant respectivement le niveau et le personnage actuel du jeu.

- ## Interfaces

L'interface CharacterFactory définit un ensemble de méthodes pour obtenir les chemins d'images liés aux actions des personnages. Trois classes, implémentant cette interface pour différents niveaux de jeu (Easy, Medium, Hard), fournissent les chemins d'images spécifiques.

L'interface mère est définie comme suit :

```
interface CharacterFactory {
   String getFireImage();
   String getShieldImage();
   String getInversedFireImage();
   String getInversedShieldImage();
}
```

La classe FireAndShieldManage utilise cette interface pour dynamiquement obtenir les chemins d'images en fonction du niveau de jeu sélectionné. Ainsi, le code reste modulaire, indépendant des détails d'implémentation, permettant une extension facile à de nouveaux niveaux sans modification substantielle du code existant.

- ## Interfaces graphiques :

Les classes relatives aux interfaces graphiques sont regroupés au sein du package "views".

- **WelcomingPage (Page d'Accueil)** : Cette classe est en charge de l'affichage initial du jeu. Elle utilise les composants Swing tels que JFrame, JLabel et JButton pour présenter les options de sélection du personnage et du niveau au joueur. De plus, elle gère les transitions vers d'autres fenêtres du jeu.

- **Labyrinth (Labyrinthe)** : La classe "Labyrinth" est responsable de l'affichage graphique du labyrinthe du jeu, ainsi que des éléments visuels tels que le personnage, le dragon et les bonus. Elle coordonne également les mouvements du joueur et du dragon, ainsi que les actions spéciales.
- **RightSidePanel (Panneau Latéral Droit)** : Cette interface graphique affiche des informations cruciales pendant le jeu, telles que la vie du joueur, la vie du dragon, le niveau actuel et le personnage choisi. Elle utilise JPanel et JLabel pour présenter ces informations de manière claire.
- **WinOrLose (Gagner ou Perdre)** : Cette classe est une fenêtre graphique qui s'affiche à la fin du jeu, indiquant si le joueur a gagné ou perdu. Elle prend en compte la vie du joueur et du dragon pour déterminer le résultat.

• Déploiement :

Pour déployer notre jeu 'Donjons & Dragons', un Makefile est fourni, facilitant son exécution. Les étapes à suivre sont les suivantes :

- Exportez l'ensemble du projet.
- Ouvrez un terminal et naviguez jusqu'au répertoire du projet.
- Entrez la commande make dans le terminal pour compiler le jeu.
- Lancez le jeu et commencez à jouer.

Le Makefile peut être ajusté selon les besoins spécifiques de l'environnement de développement.

• Conclusion

Récapitulatif des Objectifs Atteints : Ce projet a été une opportunité remarquable pour approfondir nos connaissances en Java et en développement de jeux. Nous avons réussi à créer un jeu 'Donjons et Dragons' avec une interface utilisateur riche et une logique de jeu complexe. Le projet a également renforcé notre compréhension des concepts avancés de POO et de l'architecture logicielle.

Difficultés Rencontrées et Solutions Apportées : Nous avons été confrontés à plusieurs défis, notamment dans la gestion des interactions entre l'IHM et la logique de jeu. Ces problèmes ont été résolus par une recherche approfondie et une collaboration d'équipe efficace, ce qui nous a permis d'améliorer nos compétences en résolution de problèmes.

Perspectives d'Évolution du Jeu : Pour l'avenir, nous envisageons d'enrichir le jeu en ajoutant de nouvelles fonctionnalités, telles que des niveaux supplémentaires, des options multi-joueurs, et une

amélioration continue de l'interface utilisateur. Ces développements visent à rendre le jeu encore plus captivant et à étendre notre audience.

- ## Annexes :

Lien pour les diagrammes UML :

https://lucid.app/lucidchart/968a187b-4ca0-4e2b-a3ea-e55950864633/edit?viewport_loc=-1058%2C349%2C2732%2C950%2C0_0&invitationId=inv_307e46a7-88b4-41d0-a867-a874213ea937

Sources :

1. Github
2. Stack overflow
3. Cours developpement jeux et video (Université de Lorraine)
4. Youtube
5. OpenClassroom
6. Craftpix
7. encycolorpedia