Projet de CLE : Finding Bob

Araya, Boutahlil, Delavergne, Donnart, Pineau, Vallée.
Université de Nantes

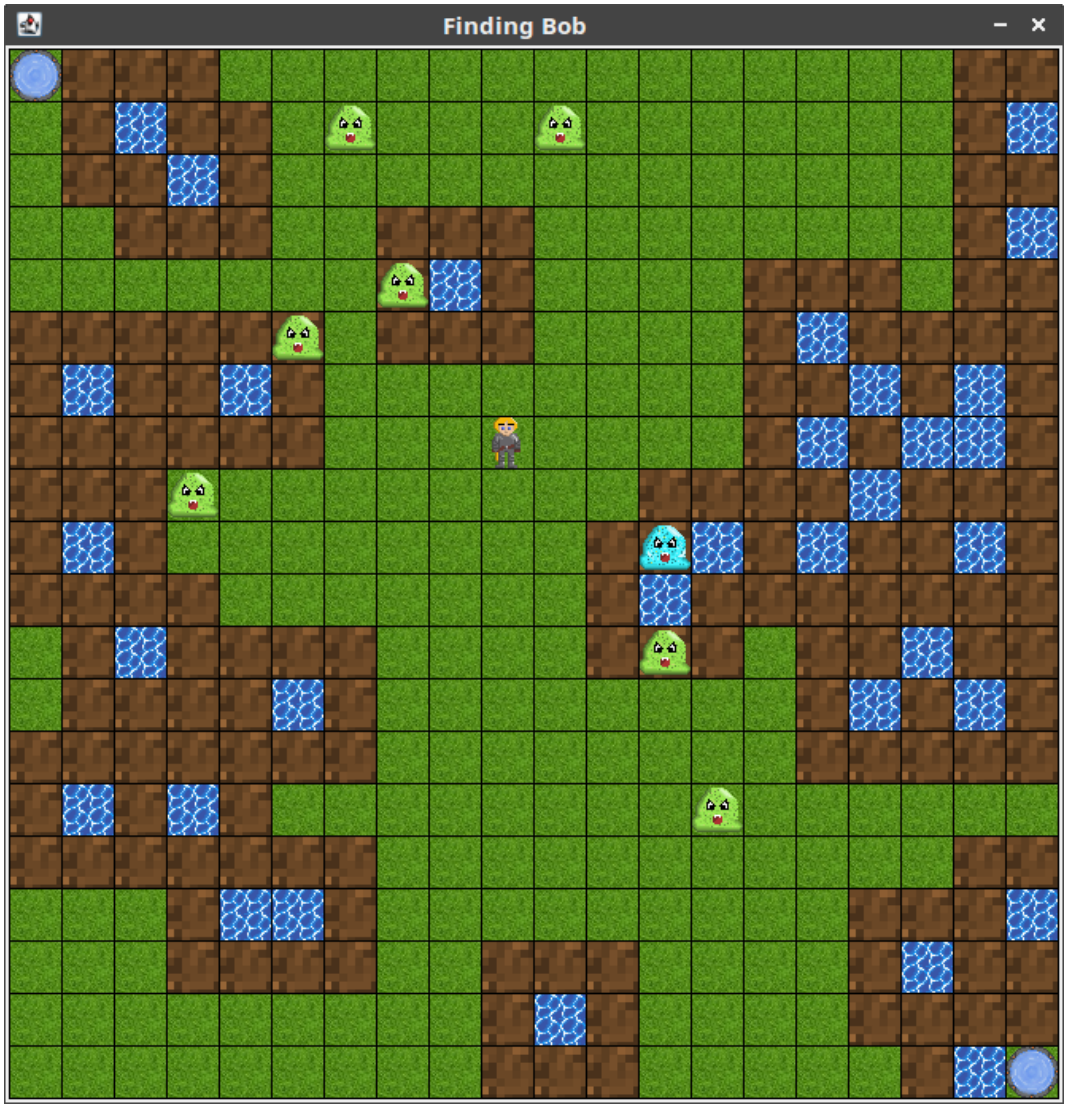

Figure 1. Jeu Principal : Finding Bob.

Table des matières

Introduction

Ce projet a pour but de réaliser une application la plus extensible possible, avec un framework utilisable par différentes applications qui viendraient se greffer dessus. Il contient séparément le framework, la partie client et des extensions (plugins) qui sont modulables pour permettre de changer entièrement le fonctionnement du projet.

1 Finding Bob c'est quoi ?

À la base, Finding Bob est un RPG basé sur un personnage qui se balade dans un monde inconnu peuplé de monstres. Le côté RPG a été finalement un peu abandonné pour mettre plus en avant le côté action, puisque le jeu permet principalement de tuer des monstres.

Le personnage se déplace avec Z, Q, S, D (respectivement HAUT, GAUCHE, BAS, DROITE). Pour combattre les monstres, il suffit de se déplacer sur eux. Bon courage !

2 Architecture dossier

Nous allons vous expliquer le rôle des différents packages. Avant de détailler le fonctionnement, voici le rôle de chacun d'entre eux :

framework contient l'intégralité des classes permettant le chargement dynamique des objets au cœur de votre application. Elle permet également le lancement automatique d'applications.

extension.extensionsConfigs dans ce package se trouve la description de l'ensemble des plugins pouvant être chargés par le framework.

configuration et client Ces deux packages sont liés. Le premier contient l'ensemble des plugins utilisés par l'application client regroupés selon différents fichiers de configuration. Le deuxième contient l'ensemble des classes utilisées par l'application client.

2.1 Framework

Dans ce package se trouve notre framework. Notre framework permet la création dynamique d'objets, la récupération et le lancement automatique d'applications.

L 'interface IExtensionDesc définit toutes les méthodes nécessaires pour la classe DescriptionExtension.

La classe ExtensionDesc implémente l'interface précédente, celle-ci permet de décrire une extension. Chaque description utilisée par le framework contient plusieurs informations telles que :

nom le nom du plugin

nomClasse le chemin vers la classe

description la description du plugin

contrainte la classe ou l'interface implémentée par le plugin

autoRun le boolean permettant de savoir si la classe doit être lancée en autorun ou non. À savoir que chaque application ne sera pas considérée comme autoRun par défaut, à moins que cette information ne soit explicitement définie.

La classe MethodAutorun crée une annotation MethodAutorun qui permet d'annoter une classe pour définir la méthode qui sera appelée si elle est lancée automatiquement par le framework.

La classe ThreadAutorun permet de créer un thread pour chaque lancement d'une application autoRun.

La classe MainFramework est le main de l'application et permet de lancer tous les Autoruns de notre application (en respectant leur priorité de lancement, un autorun ayant une priorité de 10 sera lancé avant une priorité 9, 8 ... 1).

La classe PartieProvider est le coeur de notre framework. C'est par cette classe que vous pouvez utiliser et charger dynamiquement des instances de classes à partir de votre application/plugin.

2.2 Comment utiliser notre framework

Nous n'allons pas détailler ici le fonctionnement de notre framework. Nous allons expliquer ici comment vous pouvez utiliser notre framework avec une application que vous aurez créée. Pour cela, il faut comprendre que notre framework est indépendant de toutes les autres applications. Nous allons donc imaginer que vous ne possédez que le package framework (que vous ne pourrez pas modifier) et nous allons voir deux choses : comment créer un autorun pour lancer une de vos applications personnelles et comment créer un moniteur pour le framework.

2.3 Créer une classe AutoRun

Votre classe autoRun est un plugin. Chaque plugin utilisé par notre framework doit posséder un fichier de description dans le package **extension.extensionConfigs**. Votre fichier de configuration doit ressembler à ceci :

extension/extensionConfigs/VotrePremiereApp.txt
Nom = Votre premiere application
NomClasse = chemin_vers_votre_application.VotrePremiereApp
Description = votre application trop cool
Contrainte = chemin_vers_votre_application.VotrePremiereApp
AutoRun = true

L'ensemble des informations décrites plus haut sont obligatoires. Le NomClasse et la Contrainte sont ici similaires puisque votre application ne possède aucune classe mère (à part Object de java).

Quand vous allez lancer notre framework, celui-ci va aller lire l'ensemble des fichiers de configuration, créer les descripteurs correspondants et ensuite charger par ordre de priorité les classes ayant défini leur attributs AutoRun sur "true".

Cependant, créer ce fichier n'est pas suffisant. En effet, vous devez modifier la classe de votre application pour y intégrer une annotation permettant de définir la fonction principale de votre application. Votre classe doit ressembler à celle-ci :

```
@MethodAutorun(run="lancer")
public class VotrePremiereApp{
    public void lancer(){
        //ce que vous voulez
    }
}
```

Notre framework va créer un thread, créer l'instance de votre application puis appeler la méthode définie sur cet objet. Cette méthode est obligatoire et doit être distinguée du constructeur de votre application.

Votre application autoRun et maintenant créée et vous pouvez faire ce que vous voulez avec celle-ci. Au lancement du framework, elle sera lancée en parallèle de tous les autres plugins autoRun. Nous verrons dans une autre section, en prenant exemple sur notre application client, comment vous pouvez utiliser notre framework pour charger les instances que vous souhaitez utiliser.

2.4 Créer un moniteur pour le framework

Avant de voir le client, nous allons regarder comment créer un moniteur pour le framework. Notre classe partieProvider étend la classe Observable. À chacune de ses méthodes, les évènements sont envoyés à tous ses observers sous forme de String. Ces évenements sont de la forme "EVENEMENT : CLASSE_RESPONSABLE". Ainsi, lorsque notre framework va lancer une classe AutoRun, deux évènements seront notifiés. Sur votre application précédemment créée, ces évènements sont :

- "CHARGEMENT_INSTANCE : chemin_vers_votre_application.VotrePremiereApp" lorsque votre application aura été instanciée.
- "LANCEMENT_INSTANCE : chemin_vers_votre_application.VotrePremiereApp" lorsque la méthode Run de votre application aura été lancée.

Vous devez créer une classe qui implémente l'interface Observer de java. Lorsque votre moniteur sera lancé via le framework (en autoRun), il sera directement ajouté en tant qu'observeur du framework et sera notifié à chaque évènement. Il est à noter que vous devez donc renseigner comme contrainte dans le fichier l'interface Observer.

Vous pouvez ensuite faire l'affichage (dans le terminal ou en fenêtre) que vous voulez. Il est conseillé de mettre une priorité de 10 sur votre Moniteur afin qu'il soit lancé en premier et capte le maximum d'évènements. Il suffit de rajouter dans la configuration du plugin Moniteur la ligne suivante : "Priorite = 10". Contrairement à notre application où nous n'avions rien renseigné (par défaut, la priorité est à 1), nous définissons que notre moniteur aura la priorité la plus élevée et sera lancée en premier (aléatoirement avec d'autres applications qui ont une priorité de 10).

2.5 Charger des instances via votre application

Comme nous avons crée une application client qui charge dynamiquement des instances via le framework, vous pouvez également le faire avec votre application client. Vous devez alors appeler le framework via une instance de PartieProvider. Celle-ci est accessible via la méthode static PartieProvider.getInstance(). Vous pourrez alors appeler les différentes fonctions permettant de charger des objets dynamiquement.

Les principales sont (se référer à la documentation de la classe pour plus d'explications) :

- getObjetByConfig(Class< ?> contrainte, String config) qui retourne un Object. Vous passez à cette fonction la classe mère de la classe qui va être instanciée et un fichier de la forme :

  ```
  classe = CHEMIN_VERS_LA_CLASSE
  NomAttribut = ValeurAttribut
  NomAttribut2 = ValeurAttribut2
  ```

  Le mot clef "classe" doit obligatoirement être en minuscule. Le nom de chaque attribut doit commencer par une majuscule.
  La méthode va donc créer une instance de la classe définie dans le fichier si et seulement si celle-ci implémente ou étend la contrainte donnée en paramètre de la fonction. Si l'instance est créée, pour chaque attribut renseigné dans le fichier, la méthode va appeler le set correspondant et remplir les attributs de l'objet. Celui-ci sera ensuite retourné. Si la contrainte n'est pas respectée, les méthodes renverront *null*.

- getObjetByDesc(IExtensionDesc desc) qui retourne un Object. Cette méthode fonctionne de la même manière que la précédente mais prend seulement un objet descendant de la classe IExtensionDesc. Elle va ensuite créer l'objet correspondant à cette description et le retourner.

Du côté de votre application, vous n'aurez plus alors qu'à faire un cast de cet objet.

2.6 Exemple sur notre classe client Application

Première utilisation du framework Comme c'est la première fois que nous utilisions notre framework, nous avons donc créé un package **extension.extensionsConfigs** dans notre répertoire src du projet.

Création d'une interface Nous créons une interface IPersonnage comme nous le faisons normalement en Java comme visible sur la figure 2 (page 6). Celle-ci nous servira à charger l'ensemble des personnage de notre application (ennemis comme héros).

```java
package client.interfaces;

import client.Evenement;

public interface IPersonnage {

    double getPv();

    void setPv(double pv);

    double getForce();

    void setForce(double force);

    String getNom();

    void setNom(String nom);

    void setPos(int x, int y);

    int getPosX();

    void setPosX(int x);

    int getPosY();

    void setPosY(int y);

    void deplacer(String deplacement);

    void addAction(Class<? extends IAction> cl, IAction action);

    void doAction(Class<? extends IAction> cl, Evenement evt);


    String toString();

}
```

Figure 2. Création d'une interface IPersonnage

Notre interface n'a pas besoin de fichier de description contrairement aux classes (voir ci-dessous).

Création d'une classe Premièrement nous créons une classe comme vous le faites normalement en Java. La seule règle, comme vu précédemment, est pour le get et set des attribut de notre classe qui doivent respecter la syntaxe suivante get/set+"nom de l'attribut" (comme la convention JavaBeans) avec la première lettre du nom en majuscule comme par exemple *getNom()* pour récupérer l'attribut nom. Un exemple de classe est donné dans la figure 3.

```java
import client.interfaces.IAction;

public class Personnage implements IPersonnage{

    protected double pv; // Les points de vie du perso
    protected double force;
    protected String nom;
    protected int posX = 0;
    protected int posY = 0;
    protected Map<Class<? extends IAction>,IAction> actions;


    public Personnage(){
        actions = new HashMap<Class<? extends IAction>,IAction>();
    }

    public Personnage(double pv, double force, String nom, int x, int y) {
        super();
        this.pv = pv;
        this.force = force;
        this.nom = nom;
        this.posX=x;
        this.posY=y;
    }

    /* (non-Javadoc)
     * @see client.IPersonnage#getPv()
     */
    @Override
    public double getPv() {
        return pv;
    }
```

Figure 3. Création d'une interface pour les plugins

Il nous faut ensuite créer notre fichier de configuration. Notre fichier doit contenir obligatoirement :

- classe = "chemin vers votre classe à partir du dossier 'src' du projet"
- "Nom Attribut" = "valeur attribut" avec le nom de l'attribut avec un majuscule au début pour chacun des attributs de votre classe.

Notre fichier de configuration doit être rangé dans le package **extension.extensionsConfigs**. Voici donc à quoi ressemble notre fichier de configuration visible sur la figure 4.

```
classe = extension.personnageJouable.Paladin
PosX = 1
PosY = 1
Nom = Paladin
Force = 9
Pv = 20
```

Figure 4. Exemple de fichier de configuration

Comme nous voulons que notre classe soit lancée automatiquement Nous rajoutons l'annotation "autorun" au début de la classe de la manière présentée dans la figure 5.

```
@MethodAutorun(run="lancer")
public class Application{
```

Figure 5. Autorun exemple

Charger un plugin avec le framework Maintenant que nos plugins sont créés, il faut maintenant les charger.

Pour cela Nous allons faire appel à la fonction *getObjetByConfig()*. La figure 6 montre ce que nous faisons pour nos personnage

```
IPersonnage principal = (IPersonnage) PartieProvider.getInstance().getObjetByConfig(IPersonnage.class,
        "src/configuration/configHeros.txt");
IPersonnage vilain = (IPersonnage) PartieProvider.getInstance().getObjetByConfig(IPersonnage.class,
        "src/configuration/configVilain.txt");
```

Figure 6. Chargement d'instance

Essayons de comprendre l'appel de la fonction en détaillant chaque partie :

1. Tout d'abord il vous faut faire un cast de votre objet dans le type désiré : (IPersonnage) dans l'exemple.
2. Ensuite il y a un appel statique de la fonction getObjetByConfig : *PartieProvider.getInstance().getObjetByConfig()*.
3. Les deux attribut de *getObjetByConfig()* sont :
   - "type de l'objet à créer".class
   - "chemin du fichier de configuration de cette classe"

2.7 client

Ce package contient un sous package nommé "interfaces" qui possède les différentes interfaces que le client a besoin :

- IAction gérant les actions du jeu selon ce que fait l'utilisateur,
- IAfficheur qui s'occupe de l'affichage graphique de l'application,
- IAJ qui s'occupe des actions de base qui constituent les règles du jeu,
- IEntreeUtilisateur gérant les entrées utilisateur (ce qu'il tape sur le clavier),
- IMap gérant la map du jeu,
- IPersonnage regroupant les méthodes gérant les personnages du jeu.

Ensuite dans celui-ci se trouve différentes classes :

La classe Application est un autoRun et est donc lancer automatiquement par notre framework. La fonction que le framework doit lancer est spécifier grâce à l'annotation MethodAutorun et son attribut run est la fonction "lancer". Cette classe représente le déroulement de notre jeu, tant que l'action du jeu est à true, le jeu continu de tourner.

La classe Jeu permet de définir le jeu. Un jeu possède des attributs obligatoires qui sont :

- un héros (IPersonnage),
- une map (IMap),
- un afficheur (IAfficheur),
- une entrée (IEntreeUtilisateur),
- un boolean gameOn qui permet de savoir si le jeu est terminer ou non.

La classe ChargeurPartie s'occupe de charger les différents plugin nécessaires en utilisant les méthodes du PartieProvider (framework). Cette classe possède trois méthodes public static :

- charger() permet de charger le jeu. Charge les pulgins de base pour : le personnage principal (IPersonnage), la map (IMap), l'actionDeplacer (IAction), et l'IEntreeUtilisateur.
- chargerAfficheur() permet de charger un IAfficheur qui s'occupe de la partie graphique du jeu.
- chargerActionJeu() permet de charger un IAJ (interface action jeu) qui s'occupe des actions du jeu. Il charge le plugin définit "de base" comme IAJ.

La classe Evenement s'occupe de définir plusieurs évènement : les entrées que peut faire l'utilisateur avec le clavier. Pour notre client, elle ne s'occupe pour l'instant que des déplacements : haut,bas,gauche,droite.

La classe Personnage implémente l'interface IPersonnage. Elle permet de représenter un personnage "simple" par défaut. Il possède un nom, des points de vie, de la force, une position x et y et une map d'IAction.

2.8 configuration

Dans ce package se trouve les configurations dites de "base" de l'application. Chaque fichier doit être sous la forme :

```
classe = Chemin_vers_la_classe_extension
```

Voici les six fichiers de configuration que notre application a besoin actuellement :

- "configAction.txt" -> chemin vers la classe implémentent IAJ
- "configActionDeplacer.txt" -> chemin vers la classe implémentent IAction
- "configAfficheurGraphique.txt" -> chemin vers la classe implémentent IAfficheur
- "configEntreeIhm.txt" -> chemin vers la classe implémentent IEntreeUtilisateur
- "configHeros.txt" -> configuration du héros principal
- "configMap.txt" -> chemin vers la classe implémentent IMap

2.9 extension

Dans ce package se trouve toutes les extensions/plugins que nous avons pu développer (Le jeu Finding-bob, snake, moniteur..). Afin de garder une cohérence dans le rangement de notre application, nous vous conseillons de mettre vos futurs extensions dans celui.

Le package configMap possède les différentes configurations de/des maps de nos extensions.

Le package configExtension possède les différentes configurations de nos extensions.

Le package personnageJouable implémente différents types de personnage (Mage, Paladin, Voleur).

N'oubliez pas qu'il est important que chaque plugin chargé restes compatibles ensemble. Par exemple, la map actuellement chargé (GestionnaireDeMap) necessite que l'action de déplacement appel les fonctions deplacerGauche(), deplacerDroite(), deplacerHaut() et deplacerBas() afin de pouvoir se déplacer vers différentes map grâce au portail. Bien qu'il soit possible de pouvoir déplacer directement son personnage au niveau de l'action, le changement de map ne sera alors plus possible, sauf en modifiant la map pour prendre en compte les changement de l'action.

3 Tutoriel extension SNAKE

Dans cette section, nous allons vous expliquer comment créer une extension pas à pas. Ce tutoriel va vous expliquer comment créer une extension de type jeu "Snake" grâce a notre Application/Framework.

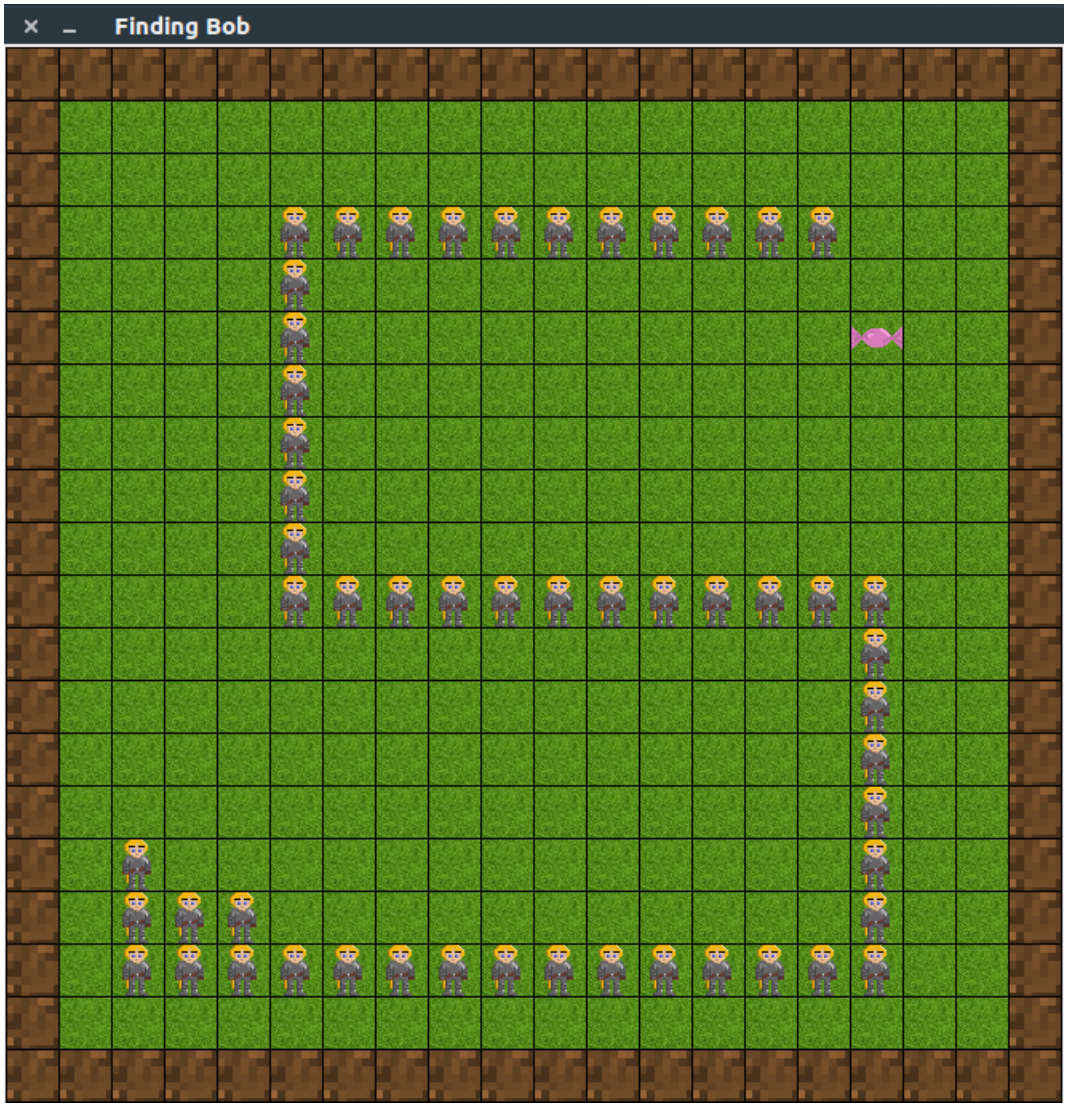

Figure 7. Extension Snake.

Ce tutoriel a pour but de vous monter que notre application n'est pas seulement orientée pour un jeu de type RPG mais ouvert à tout type de Jeu.

Tout d'abord, lorsque vous souhaitez créer votre propre extension, vous devez localiser les différentes classes que vous allez devoir implémenter ou modifier. Dans le cadre de cette démonstration, nous allons réutiliser l'interface IHM et l'afficheur du Jeu principal. Nous devrons donc créer les différentes classe qui implémentent l'interface "IMap" ainsi que l'interface "IAJ" et c'est tout !

3.1 Implémentation de l'interface "IMap"

Pour bien débuter cette implémentation, vous allez devoir localiser les différents attributs et les différentes fonctions à implémenter dans cette classe qui vous seront utiles pour le bon fonctionnement de votre extension. Dans notre exemple, nous appelons cette classe : "MapSnakeTutorial".

Attributs utiles Pour notre jeu "Snake", nous aurons besoin des attributs suivants :

IPersonnage heros représente la tête de votre serpent.

Case[[] cases ;] représente les différentes cases de votre map.

int largeur=20 ; représente la largeur de votre map.

int hauteur=20 ; représente la hauteur de votre map.

List <IPersonnage> ennemis ; représente la liste des différents bonbons, dans notre démonstration, nous n'en aurons qu'un seul.

LinkedList <Integer>corpsMonstreX ; représente le plus simplement une liste chainée des positions x du corps.

LinkedList<Integer> corpsMonstreY ; représente le plus simplement une liste chainée des positions y du corps.

boolean up = false ; représente lorsque le snake level up (mange un bonbon).

IPersonnage partieCorps ; sert d'apparence pour le corps.

Evenement dernierEvt = new Evenement(Evenement.DROITE) ; représente le déplacement par défaut (dernier déplacement initialisé à droite).

Constructeurs/méthodes/fonctions utiles Une fois que vous avez précisé tous les attributs qui vous seront utiles, vous devez implémenter le constructeur et les différentes méthodes/fonctions.

Le constructeur de notre extension IMap va initialiser :

- Ses différentes cases en "Herbe" et les contours en "Terre" (Bordure de la map),
- les deux listes chaînées à vide,
- la partieCorps en utilisant le fichier de configuration **"src/extension/configMap/-configSnakeTutorial.txt"** avec la key "queuxsnake" grâce au getObjetByConfig de notre PartieProvider,
- la liste d'ennemis en ajoutant un seul ennemi (ici un bonbon) en utilisant le fichier de configuration **"src/extension/configMap/configSnakeTutorial.txt"** avec la key "bonbonsnake" grâce au getObjetByConfig de notre PartieProvider.

Ensuite, vous allez devoir implémenter les différents Getters et Setters nécessaires et seulement implémenter les 5 méthodes *deplacerDroite(IPersonnage perso)*, *deplacerGauche (IPersonnage perso)*, *deplacerHaut(IPersonnage perso)*, *deplacerBas(IPersonnage perso)* et *removePersonnage(IPersonnage personnage)*.

removePersonnage(IPersonnage personnage) : Cette méthode sera appelée à chaque fois que le héros mangera un bonbon ; dans ce cas, cette méthode passera le boolean "up" à true et générera une nouvelle position du bonbon possible.

deplacerGauche, deplacerDroite, deplacerBas et *deplacerHaut* : Ces différentes méthodes seront implémentées quasiment de la même manière que les originales, excepté celle de droite qui sera l'appel par défaut si le joueur n'a pas saisi une nouvelle direction. Afin de savoir si cet appel est un appel par défaut ou non, nous passerons par un petit détournement en utilisant la vie du joueur. Dans notre exemple, si sa vie est égale à 99 alors cet appel est un appel par défaut et elle doit appeler la fonction appropriée de la dernière direction saisie.

À la fin de chacune de ces méthodes, nous regardons si l'utilisateur ne se trouve pas sur un mur ou sur une partie de son corps et si c'est le cas alors sa vie est mise directement à 0. Ensuite, il vous faudra actualiser la taille du corps du snake et ses différentes positions (Si up est à false alors il faut dépiler la queux des deux piles puis ajouter la position du personnage en tête des deux piles x et y). Et n'oubliez pas de modifier la dernière direction saisie à la fin de chaque fonction de déplacement.

3.2 Implémentation de l'interface "IAJ"

Dans la classe implémentant IAJ nous devrons implémenter uniquement la méthode *boolean action(Jeu j)*. Nous appellerons cette nouvelle classe : "ActionJeuSnakeTutorial". Dans celle-ci, nous devrons commencer par mettre un sleep au début de notre action. Cela permet de laisser un certain temps à l'utilisateur pour saisir une nouvelle direction ; dans notre exemple nous lui laisserons 400ms (à noter que cette durée pourrait représenter la difficulté du jeu : plus elle est courte plus le personnage avancera vite). Ensuite, vous devenez lire la saisie de l'utilisateur, si celle-ci est nulle alors passez la vie du héros à 99 et appelez la fonction *déplacerDroite* de j.getMap() ; sinon appelez *j.getHero().doAction(ActionDeplacer.class, evt)* qui se chargera d'appeler la bonne méthode de déplacement de la map.

Ensuite, vous devez regarder si le bonbon se trouve sur cette case (si *j.getMap(). getEnnemi(X, Y) !=null*) et si c'est le cas, il vous reste seulement à appeler la fonction removePersonnage (cf Constructeurs/méthodes/fonctions utiles) et appeler j.afficher().

Enfin, il ne vous reste plus qu'à regarder si la vie du héros est positive si ce n'est pas le cas, appelez j.setGameOn(false) (qui permet de mettre fin à la partie) puis n'oubliez pas de retourner j.getGameOn() ;

3.3 Configurations

Afin que l'application puisse lancer votre nouvelle extension, vous devez tout simplement modifier les fichiers de configurations nécessaires.

Vous devez :

- ajouter un fichier **src/extension/configMap/bonbonsnake.txt** sous la forme :

  ```
  classe = client.Personnage
  PosX = 9
  PosY = 9
  Nom = Bonbon
  ```

  NB : Représente le bonbon, avec une apparence data/Bonbon.png et une position $x = 9$, $y = 9$ au début.

- ajouter un fichier **src/extension/configMap/partieQueux.txt** sous la forme :

  ```
  classe = client.Personnage
  Nom = Paladin
  ```

  NB : Représente les parties du corps du serpent avec une apparence data/Paladin.png.

- ajouter un fichier **src/extension/configMap/configSnakeTutorial.txt** sous la forme :

  ```
  bonbonsnake = src/extension/configMap/bonbonsnake.txt
  queuxsnake = src/extension/configMap/partieQueux.txt
  ```

- modifier le fichier **src/configuration/configMap.txt** sous la forme :

  ```
  classe = extension.MapSnakeTutorial
  ```

- modifier le fichier **src/extension/extensionsConfigs/configActionJeu** sous la forme :

```
NomClasse = extension.ActionJeuSnakeTutorial
```

- modifier le fichier **src/configuration/configAction.txt** sous la forme :
  ```
  classe = extension.ActionJeuSnakeTutorial
  ```

3.4 Conclusion de ce tutoriel

Pour conclure, nous pouvons constater qu'il est assez trivial de créer son propre plugin/extension et en l'occurrence, notre application/framework est clairement ouvert(e) à divers types de jeux. Afin de maîtriser au maximum notre application, nous vous conseillons de refaire par vous même cette extension ou de la rendre encore plus performante (plusieurs bonbons, mettre des obstacles, enlever la limite de la map, ...). Vous pouvez également vous exercer en implémentant une extension d'un personnage (comme par exemple : un Vampire qui suce le sang de ses ennemis et donc qui régénère sa vie) ou bien implémenter un nouveau moniteur... Votre imagination est la seule limite !

NB : Vous pourrez trouver les fichiers sources de cette extension Snake dans le répertoire de notre application et de le tester. Amusez-vous bien !

Conclusion

Grâce à ce document, vous avez désormais tous les outils en main pour utiliser la plateforme avec votre propre application. Vous pouvez également améliorer notre jeu Finding-Bob grâce à votre savoir-faire. La création de cette plateforme et de Finding-Bob nous a permis de mieux comprendre la structure d'une application (framework, client, extension). Nous avons appliqué ce que nous avons appris en cours sur l'API Reflect de Java.

Table des figures