Attack on Behaviour - manuel développeur

A. Geourjon - T. Larnicol - C. Rouquier - L. Rocher - W. Dahmane - S. Chambonnet

Fonctionnement du moteur

Le moteur permet l'interaction et la mise à jour des acteurs du monde de jeu. Les acteurs du monde sont les personnages (héros, moines, guerriers, paysans). Ils peuvent interagir entre eux (combats, soins, conversions) ainsi que sur leurs environnements (récolte, attaque de bâtiment, réparation). Le moteur prévient toute anomalie et est en charge de la cohérence de l'ensemble. Par exemple, les personnages respectent un cycle de vie précis (pas de points de vie infinis ou inférieurs à 0, une case ne peut être récoltée indéfiniment, etc. Le décor est également produit par le moteur lors de la création d'une partie.

Concrètement le moteur est implémenté avec la classe Java Partie. Cette classe permet notamment de créer une partie de jeu viable et équitable lorsqu'on en construit une instance mais aussi de jouer correctement via sa méthode jouerTour().

Une partie met en oeuvre 2 types principaux d'acteurs : les héros et les hommes.

Le comportement du héros est défini par son joueur qui le contrôle librement au clavier.

Les hommes sont contrôlés par leurs automates. Le fonctionnement d'un automate est situé dans le package automate. Les deux fonctions à connaître pour les automates sont globalement: son constructeur et la méthode utiliserAutomate(List<Condition>). La méthode utiliserAutomate renvoie la prochaine action à effectuer en fonction des conditions fournies en entrée tout en mettant à jour l'état courant.

Ordonnancement

Notre ordonnancement est un peu plus complexe qu'un simple tour par tour. De par sa nature, notre jeu est à la fois un jeu d'action ou l'on contrôle un héros selon son gré et un jeu de stratégie avec les hommes et leurs automates. Par conséquent nous avons choisis de mettre à jour les personnages à chaque que l'un des héros bouge. Ainsi, on peut jouer en faisant effectuer de nombreuses actions au héros. Toutefois, si une action mérite plus ample réflexion le joueur peut prendre le temps de réfléchir et ne rien faire tout en sachant que ses hommes évolueront si le héros ennemi effectue des actions.

Ajout de contenu au moteur

Ajout de nouvelles actions

L'ajout de nouvelles actions est relativement aisé, et entraine 4 modifications du code que sont les suivantes :

- L'ajout à l'énumération Action. Notre moteur possède une énumération des actions disponibles ou l'on associe chaque action à un code (pour lire depuis un automate XML) et un décor (pour que l'automate utilisant cette action puisse être affiché sur la carte).

```
public enum Action {

  NE_RIEN_FAIRE (0, Type.ROCHER),
  ALLER_A_GAUCHE (1, Type.ARBRE),
  ALLER_A_DROITE (2, Type.SOUCHE),
  ALLER_EN_HAUT (3, Type.CHAMPS),
  \\etc..
  RECOLTER (7, Type.FERME),
  \\etc..
}
```

- Coder l'interaction du joueur pour qu'il puisse demander cette action et que l'ordonnanceur reçoive l'objet de type Action associé.

- Coder l'action en elle-même dans la classe Personnage ou Héros (par exemple : planter une fraise). Ci-après l'exemple de l'action récolter :

```
public void recolter(){
Random rand = new Random();
if(caseSousLePersonnage.getTypeDeLaCase()==Type.CHAMPS){
  proprietaire.ajouterNourriture(caseSousLePersonnage.
     recolter(recolte+rand.nextInt(recolte)));
} else if(caseSousLePersonnage.getTypeDeLaCase()==Type.ARBRE)
   {
  proprietaire.ajouterBois(caseSousLePersonnage.recolter(
     recolte+rand.nextInt(recolte)));
}
}
```

- Lier la nouvelle action à l'ordonnanceur. Pour cela rien de compliqué (et c'est même l'étape la plus simple), il suffit d'ajouter la nouvelle action au switch de l'ordonnanceur et d'écrire le code à exécuter au personnage qui est en train de joueur.

```
switch (actionAfaire) {
  case ATTAQUER:
  personnage.attaquer();
  break;
case RECOLTER:
  personnage.recolter();
```

```
            break;
        case SOIGNER:
            personnage.soigner();
            break;
        \\etc..
    }
```

Ajout de nouvelles conditions

Ce cas-ci est relativement plus complexe que le précédent. En effet les automates sont présents sur le décor et la décor de taille limité. Par conséquent, si on veut ajouter de nouvelles conditions il faudra également modifier la méthode de création de la carte afin d'avoir une carte cohérente (on ne peut pas avoir un automate de 20 par 20 alors qu'on a une carte de 40 par 40 qui doit accueillir au minimum 6 automate). A cela, il faut également effectuer 2 ajouts de code. Le premier est l'ajout de la nouvelle condition dans l'énumération Condition. Le second est plus technique et se trouve dans la méthode utiliser getAction() de la classe Homme. En effet pour utiliser l'automate on crée une liste des conditions vérifiées pour le personnage et on envoie cette liste à l'automate. Si on ajoute une nouvelle condition, il faudra écrire le code permettant de déclencher cette condition et l'ajouter à la liste des conditions à ajouter à l'automate.

Ajout de nouveaux décors

L'ajout ou la modification de décor est relativement simple. Il suffit d'ajouter le nouveau type à l'énumération, de modifier la classe Case si ce nouveau décor à des propriétés spécifiques (récoltable, bâtiment, non accessible, etc) ainsi que de mettre à disposition une image pour l'interface graphique.

Modification de la carte de jeu

La carte de jeu est générée à la construction d'un objet partie dans la méthode privée créerDecor(String nomFichierAutomates1, String nomFichierAutomates2). La carte est générée en utilisant le tableau d'actions des 6 automates du jeu ainsi que des décors comme des arbres et des champs codés en dur. Si on veut modifier le décor il faudra le coder dans la méthode créer décor de créer partie;

Modification et extension de l'interface graphique

L'interface graphique consiste en un affichage en perspective isométrique des éléments du monde à l'aide de la bibliothèque javaFX. Chaque élément du décor est représenté par une image carrée de 61*61 pixels dont certaines parties sont transparentes de façon à ce que la partie inférieure laisse apparaître un losange. Ces images en losanges sont ensuite affichées lignes par lignes en lisant la matrice du décor calculée par le moteur de jeu. Une ligne d'images

sur 2 est décalée à droite de 30 pixels afin de former un pavage du plan de forme rectangulaire. Les lignes sont affichées en commençant par les lignes du haut de sorte à ce que les éléments au premier plan recouvrent partiellement ceux du second plan, d'où l'effet de perspective.

Ajout de nouveaux décors

Il est facile d'ajouter un nouveau décor pour représenter une nouvelle action sur la carte. Pour ce faire il convient de créer une image losange de même format que les autres : prendre la forme de l'herbe pour un terrain et la remplacer par une texture du terrain en question et pour un bâtiment, prendre l'image de l'herbe et rajouter le bâtiment par dessus. Une fois l'image crée, il faut la rajouter dans le package image et ajouter une sa conversion en un type interne java dans la classe Bibliothèque. Cette classe permet de convertir les différentes images en un type interne à javaFX une seule fois (attribut final) seuls les afficheurs d'images (ImageView) sont dupliqués lors de l'affichage du décor. Ceci permet de gagner en rapidité pour l'affichage car c'est la conversion de l'image en type java qui prend le plus de temps (l'affichage prendrait environ une demi-seconde si on reconvertit l'image à chaque fois). Une fois l'image ajoutée à la Bibliothèque, il faut aller dans la classe DecorGrpahique et ajouter cette image au switch de façon à ce que les cases qui correspondent à l'image soit affichées. Pour ajouter un nouveau personnage, la démarche est similaire : créer un gif carré sur un fond transparent, l'ajouter à la bibliothèque et ajouter le cas de filtrage à la classe Personnage Graphique.

Ajout d'animations

Il serait possible d'ajouter des animations au projet. La partie la plus délicate est sans doute de dessiner l'animation et trouver un moyen de l'exporter vers un format lisible par javaFX (le format gif n'a pas forcément un bon rendu à cause du nombre limité de couleurs et de l'absence de canal alpha). Si on parvient à créer une classe animation, on peut ensuite l'instancier dans les méthodes correspondant aux différentes actions de la classe personnage. Il faut également veiller à laisser le temps aux animations de se jouer. Pour ce faire on peut faire en sorte que les tours soient cadencés pour qu'ils durent le même temps qu'une animation de déplacement d'un héros. Actuellement les tours sont joués à chaque fois qu'un joueur fait une action. Pour cela, aller dans la classe Partie et changer le handler de l'enregistreur de frappe : enlever la ligne qui fait jouer le tour et faire en sorte qu'un tour soit joué tous les x millisecondes un regardant l'heure système. Les action saisies par les joueurs étant déjà sauvegardées dans des buffers.

Afficher la carte en scrolling

Si l'on souhaite jouer avec une grande carte qui ne rentre pas sur l'écran on peut ajouter une fonctionnalité de scrolling sans difficulté. Tous les éléments graphiques du monde sont affichés dans un même noeud graphique javaFX de sorte qu'il est possible de changer le zoom et la position de l'affichage en appliquant une transformation de type changement d'échelle ou translation au noeud graphique Partie. Ainsi si l'on souhaite que le joueur ne voit qu'une petite portion

zoomée de la carte centrée sur son personnage, il suffit de zoomer la carte et la translater en fonction de la position du héros.

Génération et lecture des automates en XML

Les automates des différents personnages non joueurs sont générés à l'aide d'un petit programme ocaml qui les traduit en un fichier xml qui est lu par java. Dans ce programme on retrouve un type action et un type condition, pour ajouter une action ou une condition il faut l'ajouter dans ces types énumérés et lui attribuer un numéro dans les fonctions condition_to_int et action_to_int. Un transition d'automate est représentée pas un état initial, une condition, une priorité, une action et un état final. Un automate est représenté par le joueur auquel il appartient, un rôle (celui du personnage dont il détermine le comportement), une liste de transitions et un état initial. La traduction des automates en xml se fait à l'aide de print dans un fichier en lisant toutes les transitions. L'exécutable genère un fichier xml contenant les 3 automates pour chaque joueur. De plus la vérification de la validité des automates en faite par ce programme ocaml en appelant la fonction valider_automate qui remplace les actions interdites par Ne_rien_faire.

Architecture logicielle

Le projet étant codé en Java et donc en programmation orientée objet, nous manipulons donc des objets. Le jeu s'articule autour de l'utilisation d'un objet de classe Partie qui lui même utilise une multitude d'autres objets. Par la suite nous allons voir les principales classes du projet, leurs utilisations et leurs intérêts.

Partie

Comme nous l'avons dit précédemment la classe Partie est l'élément central de notre jeu. C'est dans cette classe que nous retrouvons le concept d'une partie de Attack on Behavior comprenant son décor, ses joueurs, etc. Nous pouvons également jouer des tours de jeu ainsi que décider si la partie est finie ou non. Les objets Partie contiennent 4 attributs principaux que nous nous permettrons de détailler ici :

- Une paire d'objets de type Joueur
- Une liste de Personnage
- Un décor composé d'un tableau à 2 dimensions de Case
- Une interface utilisateur

Il est à noter que c'est la classe Partie qui crée le décor et qui y intègre les tableaux d'actions des 6 automates du jeu (3 par joueur).

Joueur

La classe Joueur, reprend les concepts d'un joueur réel. C'est dans cette classe que l'on retrouve les quantités de ressources du joueur. La classe est responsable de la logique d'utilisation et d'accumulation de ressources. Par exemple, c'est elle qui vérifie qu'un joueur possède assez de nourriture pour acheter une nouvelle unité.

Le second point important est que Joueur en charge de la relation entre l'humain et le héros. Par conséquent c'est dans cette classe que le lien est fait entre l'action à effectuer par le héros et son joueur réel. Nous avons décidé que Joueur est une classe abstraite, ce qui nous permet de créer des joueurs particuliers afin d'enrichir le jeu. Dans notre programme nous utilisons des JoueurClavier (JoueurClavier qui est une implémentation de Joueur) Par exemple, nous pouvons créer des JoueurManette, des JoueurRéseau ou JoueurArtificiel. Le tout est que ces classes implémentent correctement Joueur.

Le troisième point important à connaitre sur les joueurs est le fait que c'est dans cette classe que sont stockés les automates originaux des personnages.

Personnage

La classe abstraite Personnage est presque aussi importante que Partie. La majorité de la logique de jeu et des actions est présente dans cette classe. Par exemple on y trouve la méthode attaquer ou recevoir dégâts par exemple. On y trouve aussi la définition des différents attributs comme la vie, la force ou les capacités de soin. Cette classe est le sommet de la hiérarchie dont tous les personnages du jeu découlent.

Heros

La classe Heros hérite de Personnage. Elle y ajoute des actions spécifiques comme créerUnité ou réparer qui sont des actions inaccessibles aux autres personnages. Des constantes spéciales y sont définies pour initialiser le Héros avec des valeurs spécifiques.

Hommes

La classe Homme hérite également de Personnage tout en y ajoutant un automate qui définit le comportement de celui-ci.

Guerriers, moines, paysans

Les classes Guerrier, Moine et Payan héritent toutes trois de Homme. Elles créent tous des Hommes différents et possédant des valeurs d'attributs spécifiques. De plus, créer 3 classes nous permet de connaitre simplement le rôle de ces Hommes.

Case

Notre décor est composé d'un tableau à 2 dimensions. Une case peut posséder un héros ou non. Une case possède également un Type. Cet attribut sera décrit dans le paragraphe suivant. Le type de la case définit les actions possibles sur cette case comme Récolter si c'est une ressource ou RecevoirDegats si c'est un bâtiment mais aussi si la case est accessible pour un personnage. Chaque case connaît sa case droite, gauche, supérieure et inférieure. le monde étant un tore les cases extérieures sont liées avec leurs homologues opposées. C'est grâce à ce principe que nous pouvons faire déplacer les Personnages. Chaque personnage connaît sa case, il lui est ainsi facile de savoir si la case à sa droite est libre et s'il peut s'y déplacer.

Type

Type est une énumération des types de décor possible. Citons arbre, champs, caserne ou Polytech.

Interface utilisateur

Automate

Dans ce projet nous avons implémenté le concept d'automate. Pour rappel, les automates sont un quintuplet d'états, d'états initiaux, finaux, de transitions et d'actions. Nous utilisons les automates afin de simuler le comportement et l'intelligence des personnages non contrôlés par le joueur.

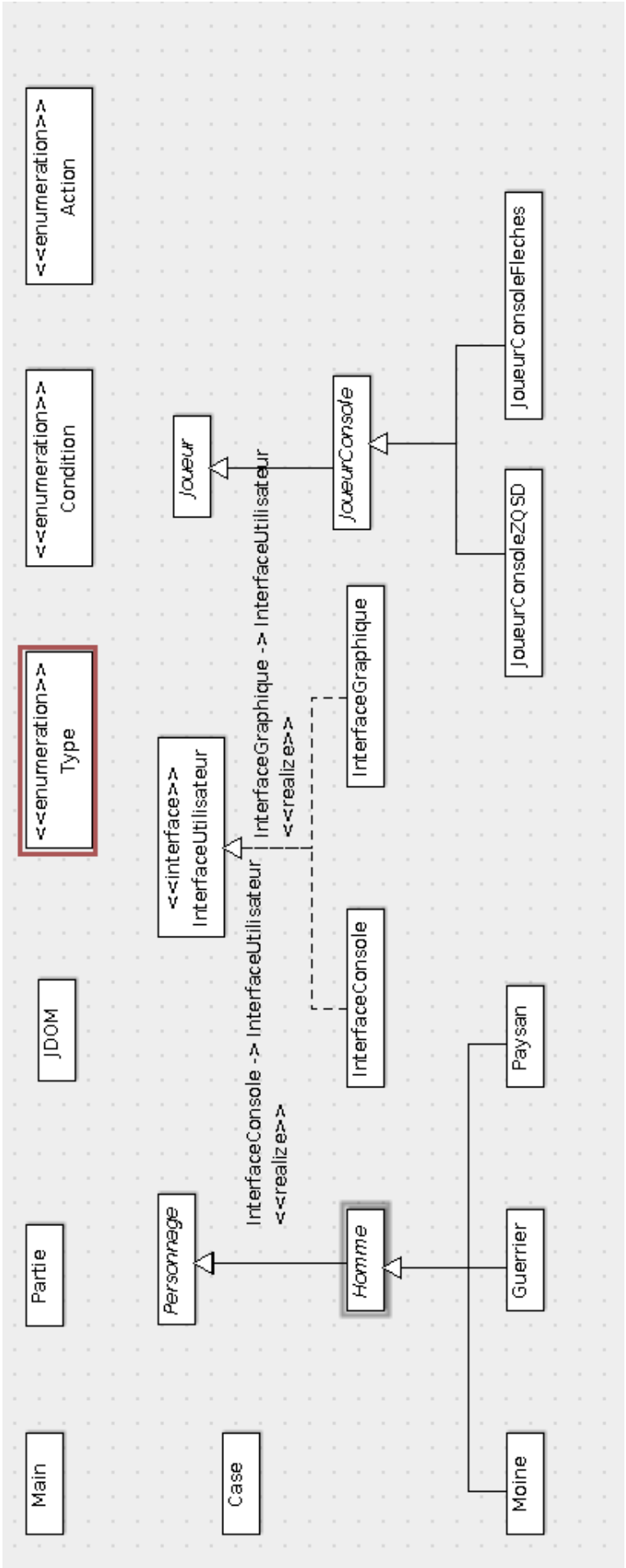
Action

Action est une énumération des actions disponibles. Citons Attaquer, récolter ou soigner. Chaque Action est associée à un type (pour être affiché sur le décor) ainsi que la traduction du code utilisé dans les automates. Définir une action de cette façon nous a permis de prévoir facilement l'extension de notre jeu tout en simplifiant la communication entre les objets du programme.

Condition

De la même façon qu'Action, Condition est une énumération des conditions d'utilisations des personnages disponibles. Citons Ennemi adjacent, Bâtiment allié adjacent ou Allié non full vie. Chaque Condition est associée à un entier pour la traduction du code utilisé dans les automates. Définir une condition de cette façon a pour but de prévoir facilement l'extension de notre jeu tout en simplifiant la communication entre les objets du programme. Les conditions prédéfinies permettent de créer des automates relativement intelligents et intéressants.

Diagramme de classe



Extrait de code

Classe automate

```
public class Automate {

  private static final int PAS_DE_TRANSIITON = -1;

  private static final int PAS_DE_PRIORITE = -1;

  private  int[] etats;

  private int role;

  private int[][] transitions;

  private int[][] priorite;

  private Case[][] actions;

  private int courant;

  private int initial;

  public Automate(int initial, int[] etats, List<Transition>
     transitionsAutomate, int role) {

    this.initial=initial;
    courant=initial;
    this.etats = etats;
    this.role = role;
    priorite=new int[Condition.values().length][etats.length];
    transitions=new int[Condition.values().length][etats.length];
    actions=new Case[Condition.values().length][etats.length];

    //Pour chaque condition
    for (int h = 0; h < Condition.values().length; h++) {
      //Pour chaque  tat  de l'automate
      for (int l = 0; l < etats.length; l++) {
        int i=0;
        while(i<transitionsAutomate.size() && (transitionsAutomate.
           get(i).getDepart()!=etats[l] || transitionsAutomate.get(i)
```

```
            .getCondition()!=Condition.values()[h])){
          i++;
        }

        //Il n'y a pas de transition pour cet  tat
        if(i==transitionsAutomate.size()){
          //On remplit avec une boucle sur soi meme et pas d'actions
          actions[h][l]=new Case(Type.HERBE, h, l);
          transitions[h][l]= PAS_DE_TRANSIITON;
          priorite[h][l]=PAS_DE_PRIORITE;
        } else {
          transitions[h][l]=transitionsAutomate.get(i).getArrivee();
          actions[h][l]=new Case(transitionsAutomate.get(i).getAction
              ().getTypeCaseAssociee(), h, l);
          priorite[h][l]=transitionsAutomate.get(i).getPriorite();
        }
      }
    }
  }


  public Action utiliserAutomate(List<Condition> conditions){

    int prioriteCourante = PAS_DE_PRIORITE;
    int conditionRetenue=0; //Initialisation inutile mais sinon il y
       a un warning

    for (Iterator<Condition> iterator = conditions.iterator();
        iterator.hasNext();) {
      Condition c = (Condition) iterator.next();
      if(priorite[c.getCodeCondition()][courant]>=prioriteCourante){
        prioriteCourante=priorite[c.getCodeCondition()][courant];
        conditionRetenue=c.getCodeCondition();
      }
    }

    //Pas de transition trouv e
    if(prioriteCourante==PAS_DE_PRIORITE){
      return Action.NE_RIEN_FAIRE;
    }else {
      Case caseDeLAction = actions[conditionRetenue][courant];
```

```
        courant=transitions[conditionRetenue][courant];
        return Action.typeToAction(caseDeLAction.getTypeDeLaCase());
      }
    }

    public int getRole(){
      return role;
    }

    public Case[][] getDecor(){
      return actions;
    }

    private Automate(int initial, int[] etats, int role, int[][]
        transitions, int[][] priorite, Case[][] actions) {
      this.etats = etats;
      this.role = role;
      this.transitions = transitions;
      this.priorite = priorite;
      this.actions = actions;
      this.courant = initial;
      this.initial = initial;
    }


    public Automate clone(){
      return new Automate(initial,etats, role, transitions, priorite,
          actions);
    }

}
```

Accès au code source et contacts de l'équipe de développement

Nous pouvons être contactés par mails :

Anthony Geourjon anthony.geourjon@e.ujf-grenoble.fr

Titouan Larnicol titouan.larnicol@e.ujf-grenoble.fr

Clément Rouquier clement.rouquier@e.ujf-grenoble.fr

Lambert Rocher lambert.rocher@e.ujf-grenoble.fr

Wallid Dahmane wallid.dahmane@e.ujf-grenoble.fr

Simon Chambonnet simon.chambonnet@e.ujf-grenoble.fr

Nous avons réalisé ce projet PLA à Polytech durant 3 semaines.