# CHATEAUTOMATE

*« La guerre des codes »*

Developer documentation

# Contents

# Introduction

Ce document est à l'intention des développeurs souhaitant étendre le projet ou l'enrichir. Il détaille notamment la structure du code, les manières de le modifier ou d'implémenter de nouvelles fonctionnalités, ainsi que quelques exemples d'implémentation native.

Nous vous conseillons vivement avant toute modification ou tentative de bien lire la section « Architecture » et de vous assurer d'en avoir bien compris l'organisation des fonctionnalités.

# Architecture

Notre architecture logicielle s'est organisée autour de deux patterns, que nous allons détailler ci-contre :

## Pattern MVC

Le logiciel est conçu selon un modèle MVC standard :

- Notre modèle est implémenté à travers le package role, cases et workshop. Il contient l'intégralité des données du jeu et assure leur persistance.
- Notre vue est gérée sous Slick2D dans le package graphique, et affiche l'interface graphique du jeu.
- Le contrôleur est un ensemble de classes contenues dans le package state qui gère l'interface entre le modèle et la vue, dans le sens où la vue requête le contrôleur afin de lire ou mettre à jour le modèle.

## Pattern Observer

Ici, ce pattern a été mise en œuvre autour des personnages et de la map. En effet, ces derniers implémentent la classe observable permettant de notifier notre système de tout événement se déroulant dans la partie graphique. Les observateurs se retrouvant du côté des contrôleurs se chargent ainsi de mettre à jour dynamiquement le modèle.

## Action & Condition

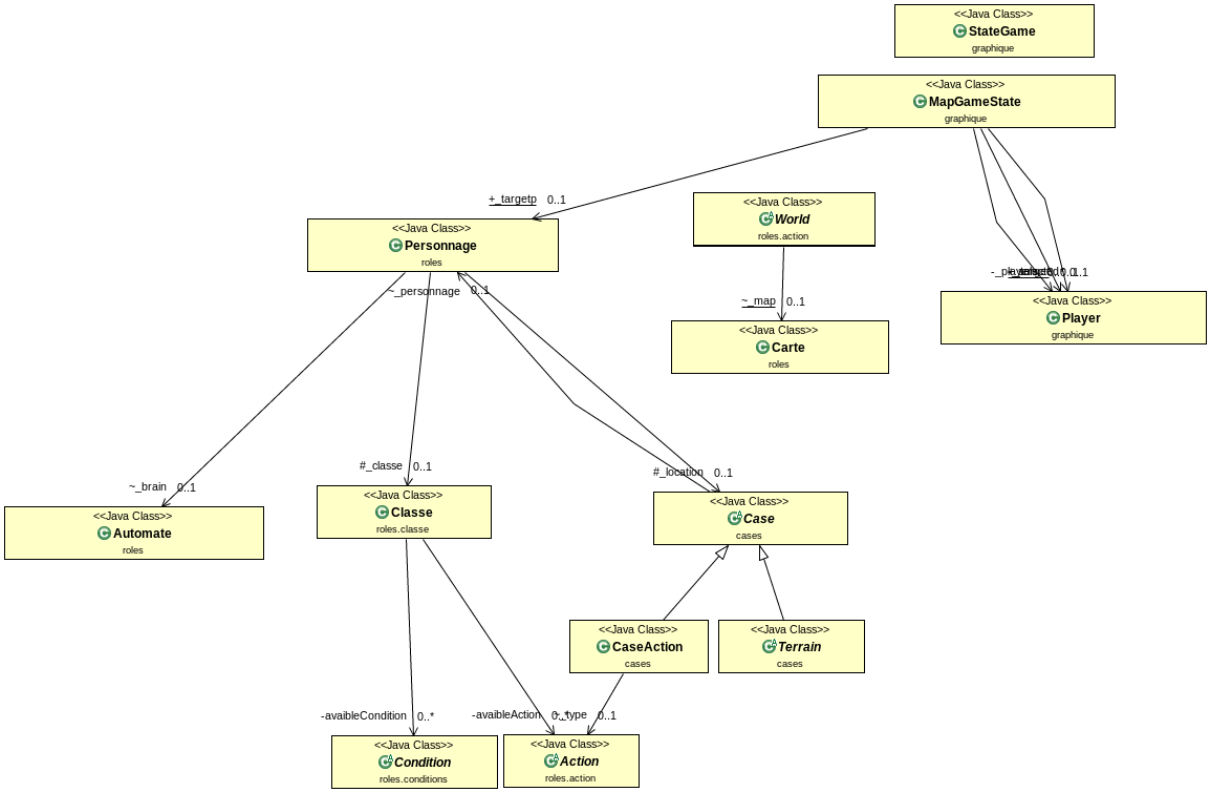
Un autre point clé de notre conception est le principe d'action et de condition régissant le jeu. Avant de vous aventurer à des modifications de contenu, veillez à bien saisir cette notion.

Les actions sont la représentation des possibilités d'action des personnages dans le jeu. Elles sont utilisées dans les automates des classes (dont sont issus les personnages) pour coder leur comportement et leur réaction, conjointement utilisés avec les conditions.

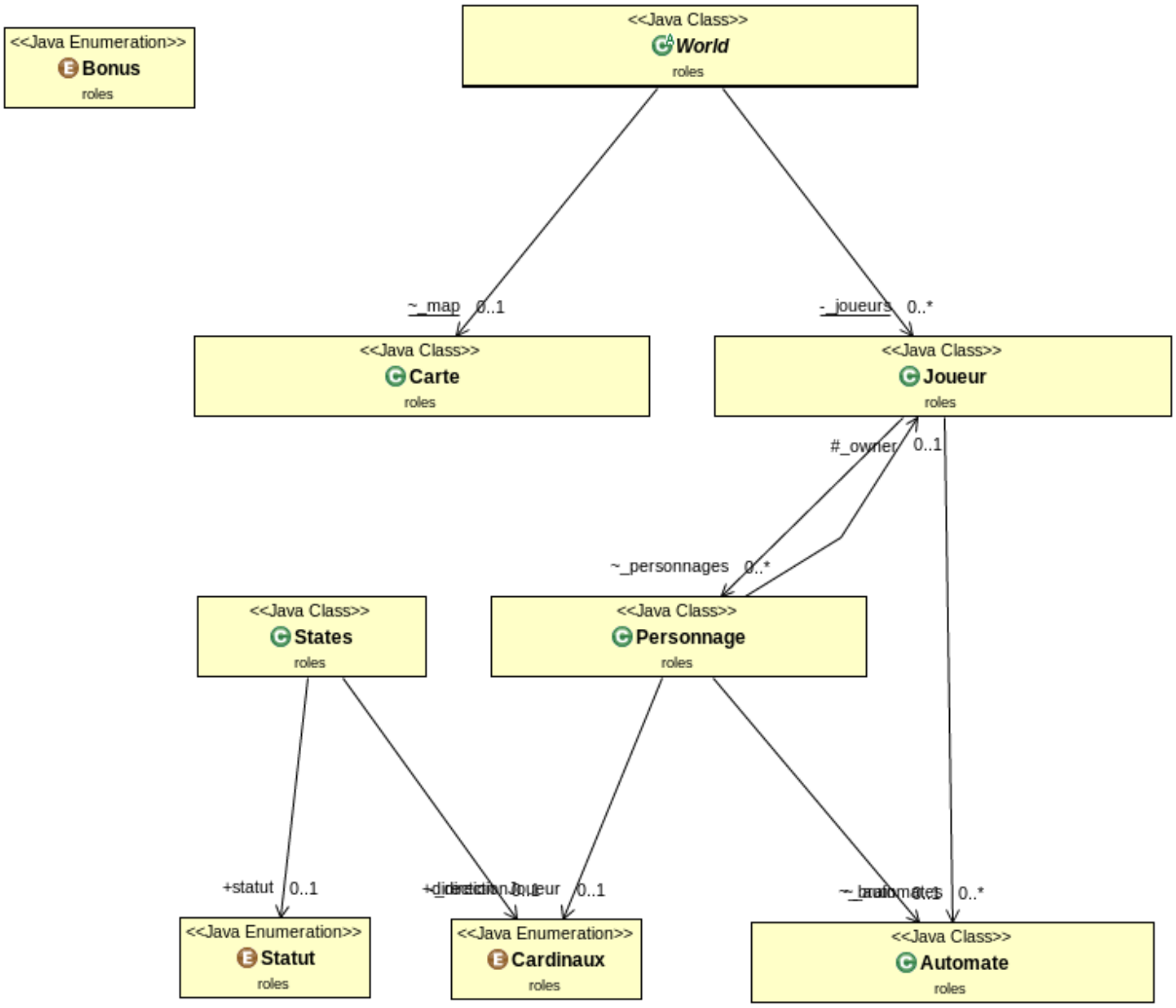
Les conditions représentent quant à elles les tests possibles dans les automates. Elles permettent donc de décider du comportement de l'automate en fonction d'un événement X. Cet événement forme donc une condition qui est implémentée pour être réutilisée dans les automates.

# Diagrammes

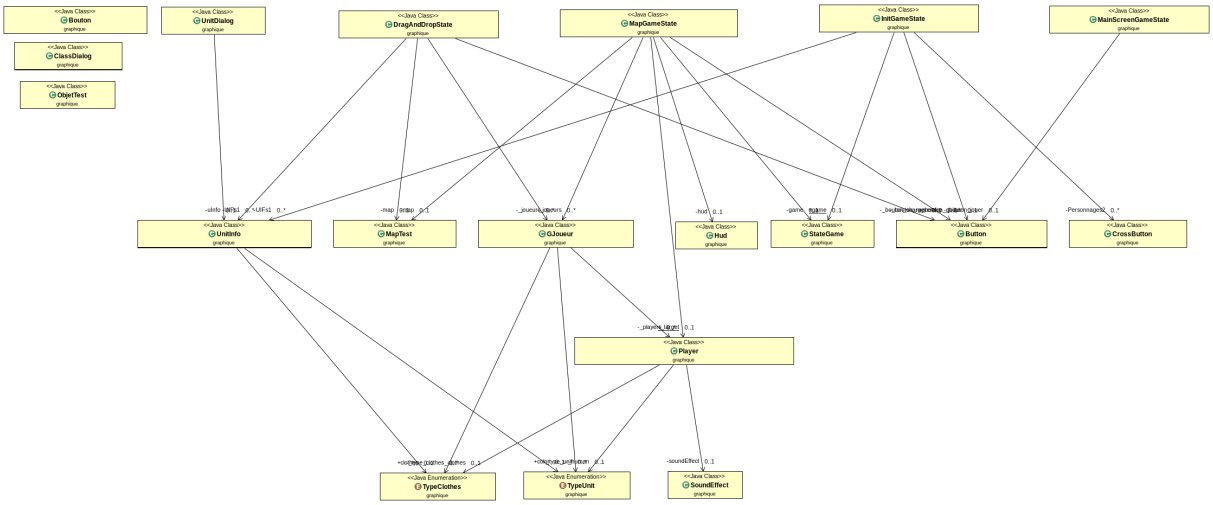
## Généralités



## Cases




## Graphique

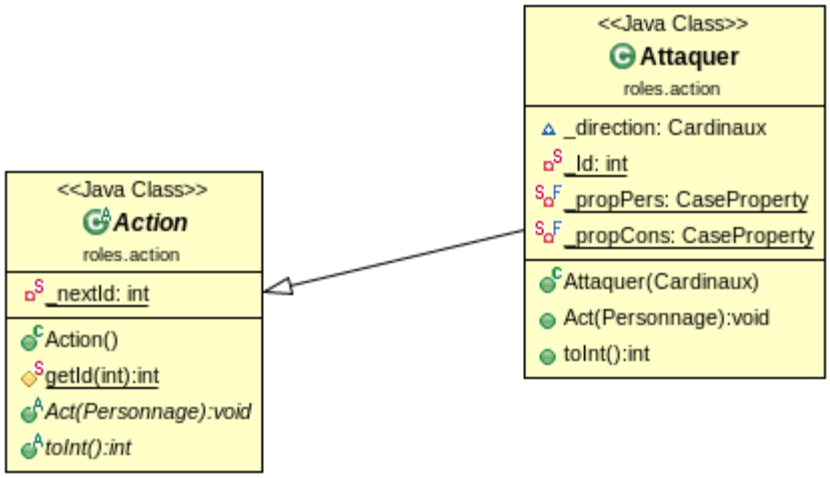


## Classe pour développeur

Les classes présentées ci-contre vous seront utiles dans toute tentative d'extension du jeu. Elles contiennent les données et les méthodes utilisées dans le jeu par nos propres implémentations.

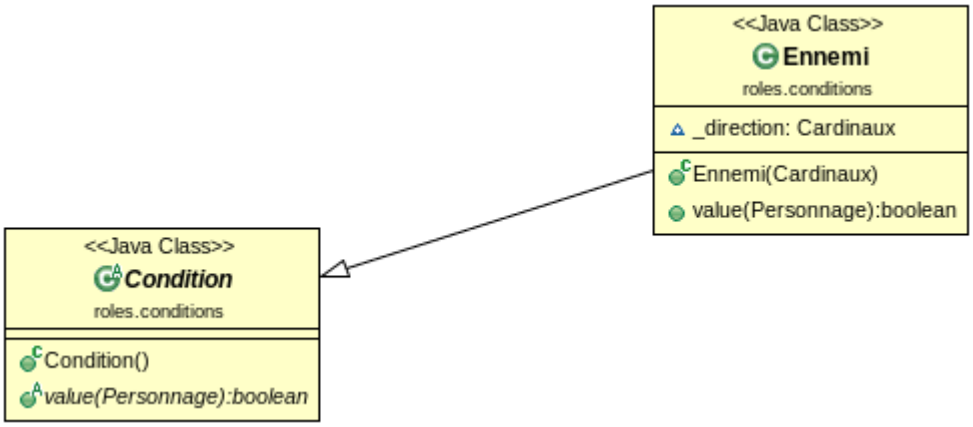
### Action & Condition

La classe Action est l'abstract générique de toutes les actions implémentées dans le jeu. Les actions correspondent aux possibilités d'action des personnages dans le jeu. Toute nouvelle action doit hériter de la classe Action.



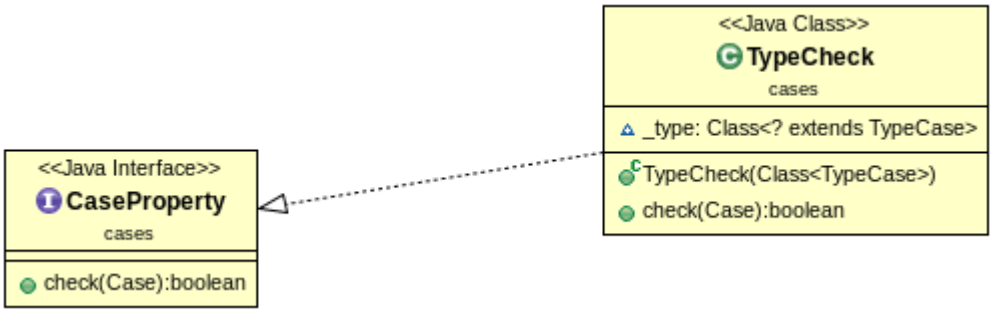

La même logique s'applique aux conditions.

On notera que les constructeurs de chaque classe implémentant Condition ou Action prennent en paramètre les attributs nécessaires à leur fonctionnement.



## CaseProperty

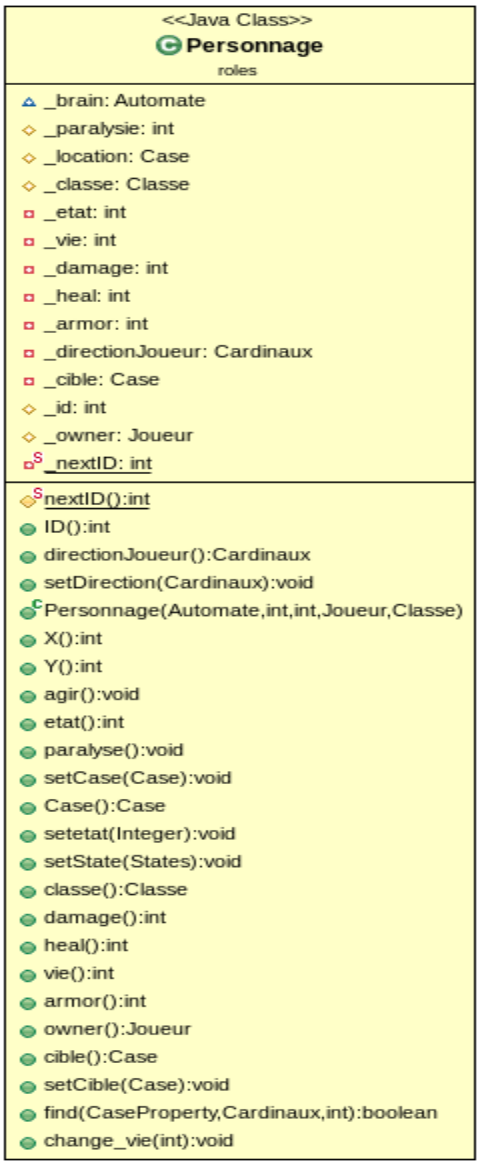
CaseProperty permet l'implémentation de la classe de test de condition portant sur les cases de la map. C'est un outil très utile dans la conception de conditions. Le fonctionnement reste le même que pour les conditions et les actions.




## Personnage

La classe personnage représente l'instance d'un personnage d'une classe donnée. Un personnage possède un comportement (son automate), une classe (dont il est issu) et une série de statistiques.

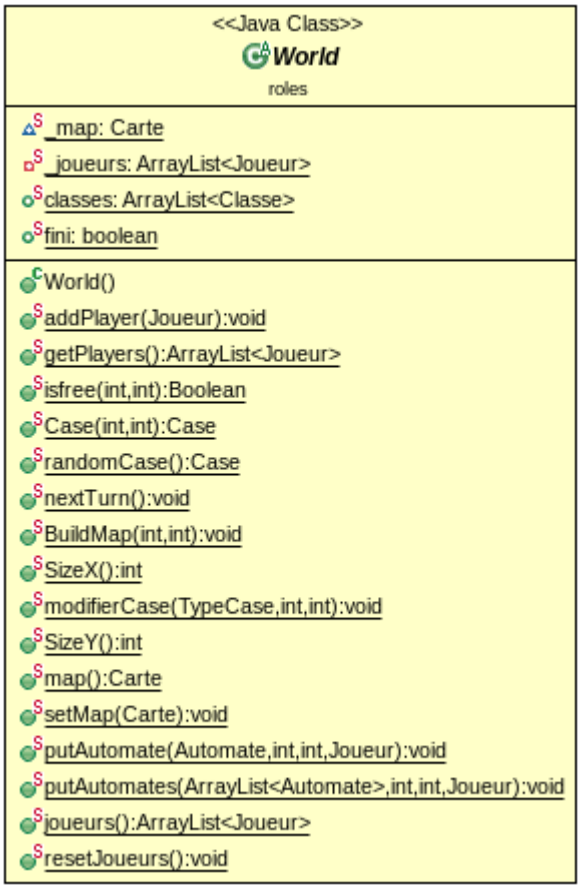
Dans le cadre d'un ajout, il est conseillé d'utiliser le personnage comme paramètre de manière à exploiter toutes les données fournies dans le modèle.



## World

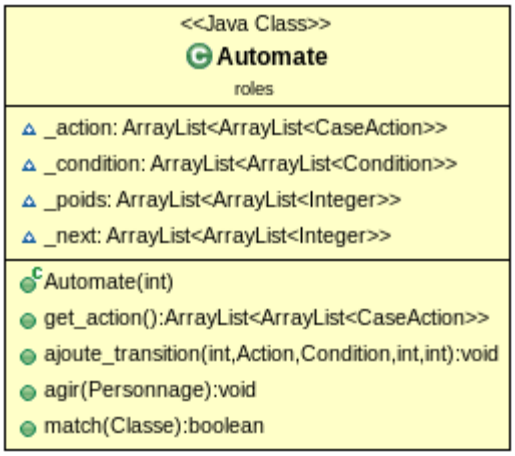
World est la classe clé de notre hiérarchie de container. Elle contient toute l'instance du jeu en terme de données (nous ne présenterons pas sa liaison avec le noyau graphique).

Elle contient donc toutes les données de la carte (qui est statique), des joueurs (et à travers eux des personnages de ce joueur), et implémente l'ordonnanceur (nextTurn) et le mapping.



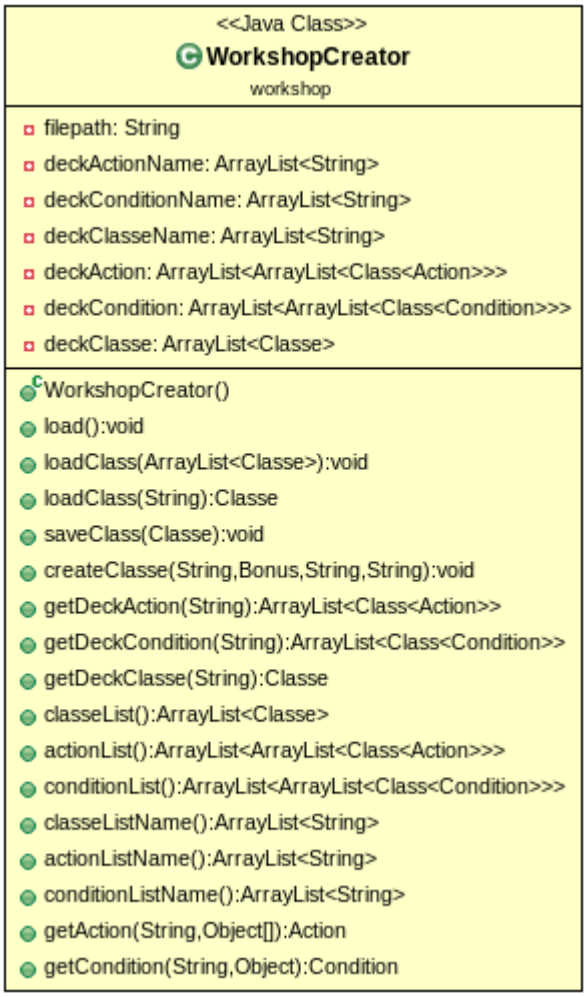

## Automate

La classe automate code le comportement des personnages dans le jeu. Elle implémente en particulier la fonction get_action qui permet d'obtenir la réaction d'un personnage.



## Workshop

Le workshop est une classe permettant l'implémentation (par le joueur) de nouveau deck de condition, d'action, et également de nouvelle classe. Elle est utilisée en une seule instance dans le logiciel, et est appelée par « StateGame.workshop ». Cette classe n'est pas un essentiel en cas d'ajout de condition ou d'action, mais toute extension portant sur les éditeurs In-Game la concernera de prêt.



# Extension du logiciel

Nous proposons un système simple d'amélioration de notre logiciel pour un développeur via notre système d'action et condition.

En effet, notre architecture permet l'implémentation de nouvelles possibilités de jeu en ajoutant de nouvelles actions réalisables par les personnages, et également de nouvelles possibilités de conditions d'actions utilisables pour définir le comportement des personnages.

## Les extensions d'action

Pour ajouter un nouveau contenu d'action, il faut :

- Créer une nouvelle classe qui doit étendre la classe abstraite Action.
- Implémenter au sein de cette classe la méthode abstraite Act(Personnage p).
- Ajouter cette action dans un automate pour l'utiliser.

A savoir :

- Le constructeur de votre action doit contenir les paramètres nécessaires à sa réalisation, tels que par exemple sa direction.
- Vous avez accès via Personnage à un certain nombre d'informations (Position, Statut, etc…), et également via les instances de World (qui permet de fournir les données sur les cartes, les personnages, des joueurs, etc…). Cela vous permet des actions relativement complexes en ayant pleinement accès à toutes les données. Pour de plus amples renseignements, se reporter à la section « Classe pour développeur ».

## Les extensions de condition

Ajouter un contenu de type condition est un peu plus exigeant que de fabriquer une nouvelle action. Il vous faudra pour cela :

- Créer une nouvelle condition qui, à l'instar d'une action, dois étendre la classe Condition.
- Implémenter la fonction abstraite value(Personnage p)dans votre nouvelle classe.
- De la même manière qu'une Action, le constructeur de votre nouvelle condition dois prendre en paramètre les objets nécessaires à sa vérification, comme sa direction.
- Si la condition porte sur la vérification d'une propriété de case, alors il vous faudra implémenter une nouvelle classe de type CaseProperty qui se chargera de vérifier la condition. En guise d'exemple, se reporter à la section « Classe pour développeur ».

# Partie OCaml

## Fichiers OCaml

Le programme OCaml permettant de créer les automates utilisés par le projet est séparé en 4 fichiers .ml différents, et 3 fichiers .mli. Chacun a une fonction particulière:

- typesAutomate.ml/typesAutomate.mli : ces 2 fichiers regroupent les types de variables utilisés dans les autres fichiers pour décrire un automate et ses différentes composantes
- constructeur.ml/constructeur.mli : définissent les fonctions utilisées par le créateur pour permettre de créer plus facilement, et plus rapidement un automate. Ils contiennent des fonctions spécifiques contenant des comportements prédéfinis, ainsi que des fonctions plus générales pour manipuler les transitions
- output.ml/output.mli : interprète un automate fourni par l'utilisateur, et l'écrit dans le fichier XML voulu. Une seule fonction est visible par le créateur: elle prend en paramètres un nom de fichier et un automate, et écrit le second dans le fichier indiqué par le premier.
- createur.ml : fichier servant d'interface à l'utilisateur, il y écrit la définition de ses automates, et ordonne au programme de produire les fichiers XML correspondant.

## Types utilisés

- cellule : une direction cardinale
- typeCellule : un type de case
- action : une action de l'automate (sortie)
- condition : une condition de l'automate (entrée)
- etat : un état de l'automate
- poids : la priorité de la transition dans l'automate
- transition : une transition définie par son état courant, la condition nécessaire à son activation, l'action accomplie, l'état d'arrivée, et sa priorité par rapport aux autres transitions
- automate : une liste de transitions, servant à décrire un automate
- attribut : nécessaire à la traduction en XML, il représente les types de paramètres d'action/condition et est associé à une chaîne de caractère grâce à une fonction de output.ml

## Structure en XML

L'automate traduit en XML est composé d'un élément "automate" possédant un attribut représentant le nombre d'états, et possédant en éléments les différentes transitions le composant.

Chaque transition possède son poids en attribut, puis chacune de ses composantes en éléments. Les actions/conditions ont leurs potentiels paramètres en attributs. Si c'est une condition composée d'une ou plusieurs autres conditions, elle a son nom en attribut, et ses composantes en sous-éléments.

## Ajout d’action/condition

Pour ajouter une action, il faut que celle-ci prenne en paramètre une direction, un type d'unité (entier), ou rien, puis dans les fichiers OCaml:

1- Ajouter l'action dans le type action de typesAutomate.ml et typesAutomate.ml

2- Ajouter le traitement XML de l'action dans output_act de output.ml

Pour ajouter une condition, il faut que celle-ci prenne en paramètre une direction, un entier, un type de cellule, un type de cellule ET une direction, 2 conditions, 1 condition, ou rien, puis dans les fichiers OCaml:

1- Ajouter la condition dans le type condition de typesAutomate.ml et typesAutomate.ml

2- Ajouter le traitement XML de la condition dans output_cond de output.ml

# Code Exemple

## Automate :

```
public class Automate implements Serializable{
	ArrayList<ArrayList<CaseAction>> _action;
	ArrayList<ArrayList<Condition>> _condition;
	ArrayList<ArrayList<Integer>> _poids;
	ArrayList<ArrayList<Integer>> _next;

	public Automate(int nb_etat)
	{
		_action = new ArrayList<ArrayList<CaseAction>>();
		_condition = new ArrayList<ArrayList<Condition>>();
		_poids = new ArrayList<ArrayList<Integer>>();
		_next = new ArrayList<ArrayList<Integer>>();
		while(nb_etat>0)
		{
			_action.add(new ArrayList<CaseAction>());
			_condition.add(new ArrayList<Condition>());
			_poids.add(new ArrayList<Integer>());
			_next.add(new ArrayList<Integer>());
			nb_etat--;
		}
	}


	public ArrayList<ArrayList<CaseAction>> get_action() {
		return _action;
	}


	public void ajoute_transition(int etat, Action a, Condition c, int etat_suivant, int
poids)
	{
		_action.get(etat).add(new CaseAction(new Batiment(a)));
		_condition.get(etat).add(c);
		_poids.get(etat).add(poids);
		_next.get(etat).add(etat_suivant);
	}

	public void agir(Personnage pers) {
		ArrayList<Integer> choice = new ArrayList<Integer>();
		for(int id = _condition.get(pers.etat()).size()-1; id >= 0; id--)
			if(_condition.get(pers.etat()).get(id).value(pers))
				choice.add(id);

		//le personnage ne fait rien, penser Ã  Attendre quand cela sera implÃ©mentÃ©
		if(choice.size() == 0)
		{
			pers.paralyse();
			pers.setState(new States(Statut.ATTENDS, Cardinaux.NORD));
			return;
		}

		//comparateur utilisÃ© pour comparer en utilisant les poids des transitions
		Collections.sort(choice, (new Comparator<Integer>() {
			int _etat;
			@Override
			public int compare(Integer i1, Integer i2)
		    {
				return  _poids.get(_etat).get(i1) - _poids.get(_etat).get(i2);
		    }
```

```
                        Comparator<Integer> init(int etat)
                        {
                                _etat = etat;
                                return this;
                        }
                }).init(pers.etat()));

                int poids = _poids.get(pers.etat()).get(choice.get(choice.size()-1));
                /*for(Integer id : choice)
                {
                        if(poids != _action.get(pers.etat()).get(id).poids())
                                choice.remove(id);
                }*/
                for(int id = choice.size()-1; id>=0; id--){
                        if(poids != _poids.get(pers.etat()).get(choice.get(id)))
                                choice.remove(id);
                }

                Collections.shuffle(choice);
                //System.out.println("Je vais " +
_action.get(pers.etat()).get(choice.get(0)).action().getClass().getName());
                _action.get(pers.etat()).get(choice.get(0)).Act(pers);
                pers.setetat(_next.get(pers.etat()).get(choice.get(0)));
        }

        public boolean match(Classe classe){
                for(int i=0;i<_action.size();i++){
                        for(int j=0;j<_action.get(i).size();j++){
                                if(!classe.isAction(_action.get(i).get(j).action().getClass())){
                                        return false;
                                }
                        }
                }
                for(int i=0;i<_condition.size();i++){
                        for(int j=0;j<_condition.get(i).size();j++){
                                if(!classe.isCondition(_condition.get(i).get(j).getClass())){
                                        return false;
                                }
                        }
                }

                return true;
        }
}
```

## World :

```
public abstract class World {

	static Carte _map;
	private static ArrayList<Joueur> _joueurs = new ArrayList<Joueur>();
	public static ArrayList<Classe> classes = new ArrayList<Classe>();
	public static boolean fini = false;

	public static void addPlayer(Joueur j)
	{
		joueurs().add(j);
	}

	// final return...
	public static ArrayList<Joueur> getPlayers()
	{
		return joueurs();
	}

	public static Boolean isfree(int x, int y) {
		return _map.isfree(x, y);
	}

	public static Case Case(int x, int y) {
		return _map.Case(x, y);
	}

	public static Case randomCase(){
		Random R = new Random();
		int x = R.nextInt(_map.largeur());
		int y = R.nextInt(_map.hauteur());
		return Case(x,y);
	}

	public static void nextTurn()
	{
		ArrayList<Joueur> vaincus = new ArrayList<Joueur>();
		ArrayList<Personnage> activated = new ArrayList<Personnage>();
		for(Joueur j : joueurs()){
			if(j.getPersonnages().isEmpty()){
				vaincus.add(j);
				System.out.print(j.nom()+" a perdu!");
			}
			else
				for(Personnage p : j.getPersonnages() )
					activated.add(p);
		}

		for(Joueur j : vaincus){
			joueurs().remove(j);
		}
		if(joueurs().size()==1){
			fini = true;
		}

		Collections.shuffle(activated);
		for(Personnage p : activated)
			if(p.vie()>0)
				p.agir();
	}

	public static void BuildMap(int hauteur, int largeur) {
		_map = new Carte(hauteur,largeur);
	}

	public static int SizeX() {
		return _map.get(0).size();
	}
```

```
	public static void modifierCase(TypeCase type, int x, int y){
		_map.modifierCase(type, x, y);
	}

	public static int SizeY() {
		return _map.size();
	}

	public static Carte map() {
		return _map;
	}

	public static void setMap(Carte map) {
		_map = map;
	}

	public static void putAutomate(Automate a, int x, int y, Joueur j) throws Exception{
		_map.putAutomate(a, x, y, j);
	}

	public static void putAutomates(ArrayList<Automate> a, int x, int y, Joueur j) throws
Exception{
		_map.putAutomates(a, x, y, j);
	}

	public static ArrayList<Joueur> joueurs() {
		return _joueurs;
	}

	public static void resetJoueurs() {
		_joueurs.clear();
	}

}
```

# Action & Exemple :

## Action :

```
public abstract class Action implements Serializable{

	private static int _nextId = 0;

	// @ensure range>0
	protected static int getId(int range)
	{
		_nextId += range;
		return _nextId - range;
	}

	public abstract void Act(Personnage pers);

	public abstract int toInt() ;
}
```

## Avancer (extend Action) :

```
public final class Avancer extends Action {

	Cardinaux _direction;
	private static int _Id = Action.getId(1);

	public Avancer(Cardinaux card) {
		super();
		_direction = card;
	}

	@Override
	public void Act(Personnage pers) {
		CaseProperty p = new LibreCheck(pers);
		int destX = pers.X() + ((_direction == Cardinaux.OUEST)? (-1) : ((_direction ==
Cardinaux.EST)? 1 : 0));
		int destY = pers.Y() + ((_direction == Cardinaux.NORD)? (-1) : ((_direction ==
Cardinaux.SUD)? 1 : 0));
		if(p.check(World.Case(destX, destY)))
		{
			if(World.Case(destX, destY).type() instanceof Arbre )
			{
				//System.out.print(pers.ID() + "j'avance vers l'arbre " +
_direction + destX + " " + destY + ".\n");
				World.Case(destX, destY).setPersonnage(pers);
				pers.setState(new States(Statut.HIDING, _direction));
			}
			else if(World.Case(pers.X(), pers.Y()).type() instanceof Arbre)
			{
				//System.out.print(pers.ID() + "j'avance vers le " + _direction +
destX + destY + ".\n");
				World.Case(destX, destY).setPersonnage(pers);
				pers.setState(new States(Statut.REVEAL, _direction));
			}
			else
			{
				//System.out.print(pers.ID() + "j'avance vers le " + _direction +
destX + " " + destY + ".\n");
				World.Case(destX, destY).setPersonnage(pers);
				pers.setState(new States(Statut.AVANCE, _direction));
			}
		}
	}

	@Override
	public int toInt() {
		return _Id;
	}
```