Rapport de projet
Projet Long de Java

Groupe 34
Lorraine PERRONNET - Hassan KHATIB - Audric SCHILTKNECHT

Juin 2006

Résumé

Simulateur de diagramme d'état

Table des matières

1 Analyse

1.1 Introduction

Le but de ce projet est de réaliser un simulateur de diagramme d'état UML. On devra afficher les évènements produits, et les interactions qui en découlent.

De plus, des interfaces textuelle et graphique seront réalisées.

1.2 Modélisation UML

1.3 Cas d'utilisation

1.3.1 Diagramme

Figure 1 se trouve le diagramme des cas d'utilisations.

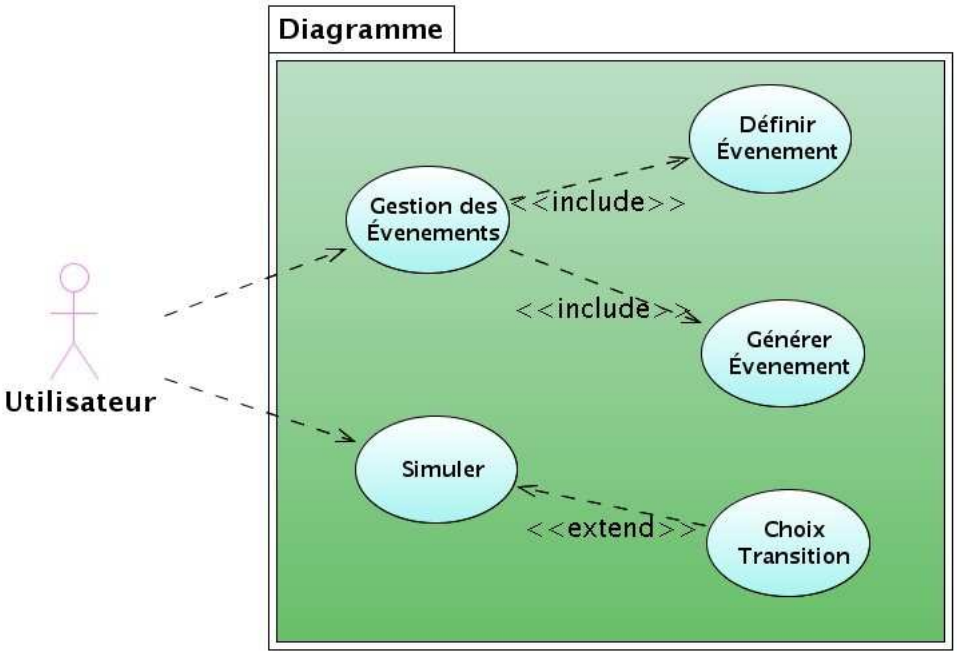

FIG. 1 – Cas d'utilisation

1.3.2 Description des cas d'utilisation

Simuler

But : Faire évoluer le diagramme d'un pas de simulation.

Début : L'utilisateur lance la simulation

Fin : Le système est à l'état $t+1$.

Description :
- Déterminer l'ensemble des transitions franchissables et cohérentes
- Franchir cet ensemble de transitions
- Effacer la liste des évènements
- Mémoriser les évènements générés pendant le franchissment des transitions

Cas alternatifs : En cas d'indeterminisme, choisir l'ensemble des transitions à franchir.

Définir un évènement

But : Créer la possibilité pour l'utilisateur de rajouter un évènement à la liste des évènements à traiter pour le prochain pas de temps.

Début : L'utilisateur commence la définiton du nouvel évènement

Fin : L'évènement a été ajouté à la liste des évènements définis par l'utilisateur

Description : L'utilisateur entre le nom de l'évènement

Générer un évènement

But : Ajouter un évènement défini par l'utilisateur dans la liste des évènements à traiter au prochain pas de temps.

Début : L'utilisateur sélectionne l'évènement à émettre

Fin : L'évènement est rajouté à la liste des évènements à traiter.

Description :
- L'utilisateur sélectionne le ou les évènement(s) à emettre.
- Les évènements sélectionnés sont ajoutés pour traitement au système.

Cas alternatifs : L'utilisateur demande de générer un évènement qui n'existe pas.

1.4 Diagramme de classe

Figure 2 se trouve le diagramme des clases de notre application.

1.4.1 Diagramme

1.4.2 Explications

On définit ainsi une classe système qui contiendra le diagramme « principal », ainsi que la liste des évènements.

Cependant, nous nous sommes rendus compte qu'à partir de l'état initial, il était possible de retrouver tous les états du diagramme. En conséquence, la classe diagramme ne contiendra donc que l'état initial, ainsi qu'une poignée sur l'état actif.

Un état initial sera représenté par une classe particulière qui contiendra les actions à effectuer lors de l'initialisation du diagramme.

Remarque : On aurait pu aussi utiliser une transition comme « état », qui contiendrait la liste des opérations correspondant au code du constructeur.

EtatAbstrait et classes dérivées La classe abstraite EtatAbstrait permet de factoriser certaines méthodes pour les classes Etat et Diagramme qui en dérivent. La classe Etat permet de représenter un état élémentaire. La classe Diagramme permet de représenter les diagrammes ET (composés de plusieurs objets du type EtatAbstrait qui évoluent simultanément) ou bien les super-états (qui contiennent plusieurs objets du type EtatAbstrait, un seul de ces objets peut être actifs à la fois). L'attribut `etatsInitiaux` de la classe Diagramme permet de représenter les diagrammes ET. L'état initial qui permet d'entrer dans le diagramme n'est pas visible du diagramme. C'est toujours les diagrammes englobants qui gèrent cette transition. C'est pourquoi sur les diagrammes de test on ne représente pas l'état initial le plus extérieur au système.

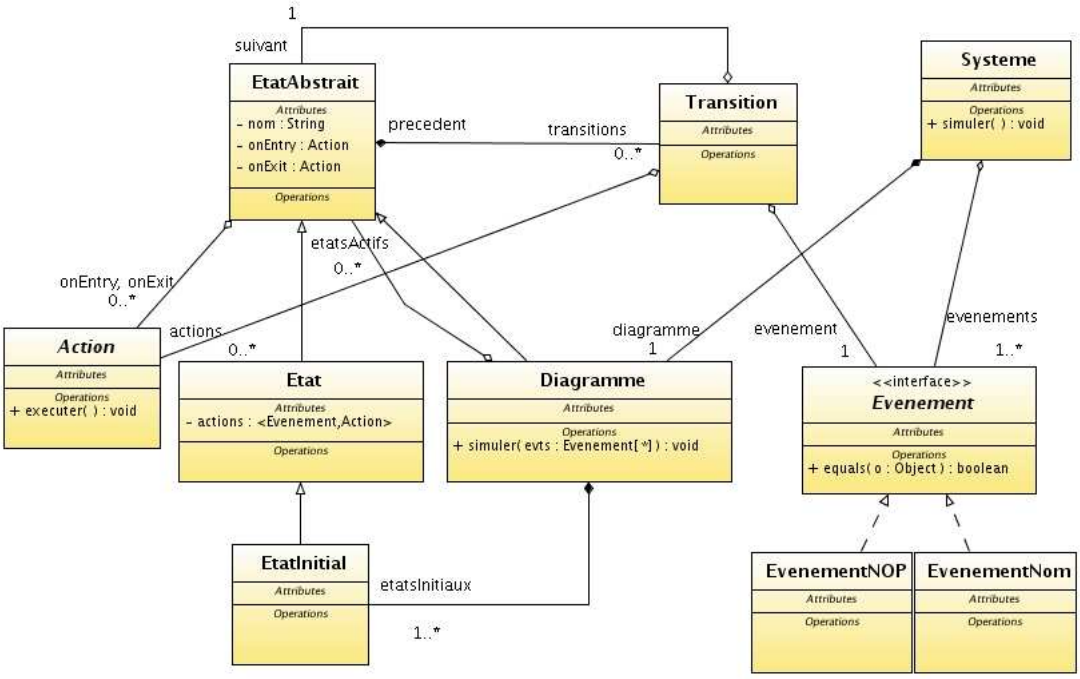

FIG. 2 – Diagramme de Classe

1.5 Algorithme

À un instant donné, on souhaite effectuer un pas de simulation. Pour cela, on va parcourir la liste des états actifs du diagramme « principal ».

Pour chacun de ces états actifs, on vérifie s'il existe une transition franchissable :

Oui On franchit la transition

Non On appelle la fonction de simulation sur l'état

Si l'état est un diagramme, alors la fonction de simulation est celle que l'on vient de décrire. Si c'est un état « normal », alors on se contente de vérifier s'il existe des actions internes que l'on peut exécuter.

Remarque : En cas d'indeterminisme, on demandera à l'utilisateur la transition à franchir.

2 Ergonomie des interfaces graphiques

2.1 Interface pour le clignotant

La figure 3 présente l'ergonomie de cette interface.

L'utilisateur dispose de boutons « radio » lui permettant de sélectionner exclusivement l'un des trois état : *gauche*, *droite*, *warning*. Il peut ensuite faire évoluer la simulation d'un pas, et observer l'état des feux à l'aide de la visualisation.

FIG. 3 – Ergonomie de l'interface Graphique - Clignotants

3 Interface pour l'application générale

3.1 Gestion du diagramme

Cette partie de l'interface est présentée figure 4.

Cette partie de l'interface graphique permet de gérer l'état du diagramme. Y sont représentés les divers états, ainsi que ceux qui sont actifs, les transitions franchies, la liste des évènements à traiter, ainsi que la possibilité pour l'utilisateur de générer un ou plusieurs évènements à sa demande.

FIG. 4 – Ergonomie de l'interface Graphique - Gestion du diagramme

3.2 Gestion des évènements

La figure 5 représente la partie permettant à l'utilisateur de gérer des évènements qu'il pourra générer au cours de l'utilisation. Il peut ainsi créer un évènement, le modifier, ou tout simplement le supprimer de la liste.

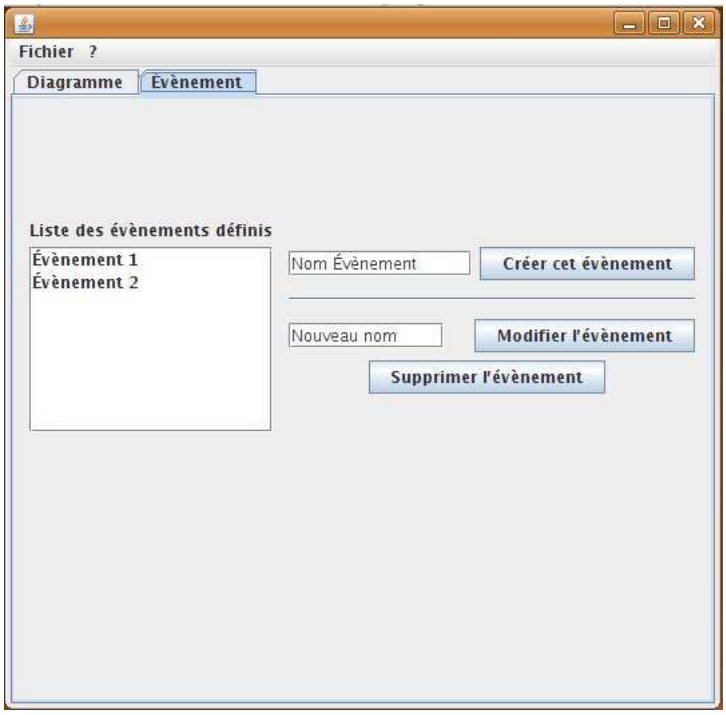

FIG. 5 – Ergonomie de l'interface Graphique - Gestion des Évènements

3.3 Répartition du travail

Le travail a été distribué de la façon suivante :
- Lorraine se chargera du modèle et de l'onglet de gestion des évènements
- Hassan travaillera sur la vue spécifique, et de son intégration avec le simulateur
- Audric codera une partie du modèle avec Lorraine, ainsi que l'onglet de gestion du diagramme

4 Conception - Implémentation

Nous allons décrire ici les différents choix réalisés dans la conception et l'implémantation du projet

4.1 Application du design pattern Modèle - Vue - Contrôleur

Puisque l'on doit réaliser une interface graphique, nous avons décidé d'appliquer le *design pattern* Modèle - Vue - Contrôleur.

Le modèle représentera le « simulateur » en lui-même : il disposera de méthodes permettant d'effectuer la simulation, mais ne proposera aucun moyen de visualiser son état interne.

Pour que l'utilisateur puisse avoir un retour de l'état du modèle, nous utiliserons le patron de conception Observateur : un diagramme d'état possèdera un ensemble d'observateurs qui seront avertis à certains moment clés de la simulation : activation d'un état, franchissement d'une transition, ...

La vue sera en fait une réalisation de l'observateur. Ainsi, elle sera averti des modifications de l'état interne du diagramme, et pourra donc être modifiée en conséquence.

Quant au contrôleur, son rôle est de servir d'interface entre les actions effectuées par l'utilisateur sur la vue, et le modèle. Ainsi, lorsque l'utilisateur voudra ajouter un évènement au système, la vue notifiera le contrôleur de cet ajout, qui lui même appellera la méthode adapté du système.

Remarque : La vue étant développée sous l'application `Netbeans`, la vue et le contrôleur seront étroitement liés.

4.2 Système

L'appel au constructeur du Système initialise aussi le diagramme associé (cela évite ainsi les "oublis" de la part de l'utilisateur.

Cette classe contiendra la liste des évènements créés lors du pas de simulation précédent, ainsi que la liste des évènements qui ont été ajouté lors du pas de simulation courant par les diverses actions.

On stockera aussi dans cette classe le nombre de pas de simulation exécutés depuis l'initialisation du diagramme.

4.3 Évènements

Un évènement étant décrit par son nom, une simple classe pouvait suffire.

Néanmoins, on peut disposer dans le diagramme de transition sans évènement associé. Nous avons donc décidé, au lieu d'affecter l'évènement d'une telle transition la valeur `null`, de créer une classe `EvenementNOP`, qui représentera l'évènement « vide ». Cet évènement sera systématiquement ajouté à l'ensemble des évènements du système.

Ainsi, il ressort que la classe `EvenementNOP` sera extrémement utilisée, puisque ajouté par défaut à toutes les transitions qui ne possèdent pas d'évènements, ainsi qu'au système. Nous avons donc décidé d'utiliser un mécanisme de classe singletone : une seule instance de cette classe sera créée et utilisée tout au long de l'éxecution du programme.

Pour cela, il faut utiliser le mécanisme suivant :
- Déclarer un attribut du type `EvenementNOP`
- Créer un constructeur de visibilité `private`
- Créer une méthode d'accès statique : c'est cette méthode qui devra être appellée en lieu et place du constructeur.

La méthode utilisée pour la construction est la suivante : il faut regarder si l'attribut a déjà été initialisé :

Oui : On retourne l'attribut

Non : On initialise l'attribut, par appel au constructeur privé, et on le retourne

4.4 Transition

Une transition sera construite à partir de l'état vers lequel elle amène. On pourra lui ajouter un évènement, qui sera la condition du franchissement de la transition.

Il est aussi possible d'ajouter une ou plusieurs actions qui seront exécutées lors du franchissement de la transition.

La principale méthode de la classe `Transition` est la méthode `franchir`. L'appel à cette méthode suppose que l'on ait au préalable vérifié que l'évènement de la transition soit bien présent dans le système.
La méthode prend en paramètre la liste des états actifs du système.

Les étapes de cette méthode sont les suivantes :
- Execution des actions associées à la transition
- Activation de l'état suivant
- Ajout de l'état suivant à la liste des états actifs du système

4.5 Observateur

4.5.1 Description

Une interface `Observateur` a été définie. Elle servira à définir les méthodes qui devront être implémantées par les différentes vues, et qui seront appelées à divers instant de la simulation par le diagramme.

Pour que l'on puisse disposer de plusieures vue en simultané, nottament de l'affichage avec une vue graphique, et d'une trace console, il a fallu définir une classe qui implémente l'interface `Observateur` et représente un groupe d'observateurs. Cette classe agrège plusieurs observateurs, et leur transmet les appels aux diverses fonctions de l'interface.

On voit bien qu'un problème apparaît : en cas d'indeterminisme, quel observateur faudra-t-il choisir de notifier ?
Pour cela, on décidera que le premier observateur construit aura la priorité sur les autres. Si cela ne convient pas, une méthode `setObservateurPrincipal` est disponible.

4.5.2 Intégration avec le système

Ainsi, le diagramme contiendra un groupe d'observateurs. Cet attribut sera défini en tant qu'attribut `protected static` de la classe `EtatAbstrait`, pour que tous les états du diagramme aient accès à ce groupe.
Ainsi, il suffit dans les classes qui héritent de `EtatAbstrait` de faire un simple appel aux méthodes de l'observateurs, pour notifier tous les observateurs qui sont inscrits d'un changement.

4.6 Interface Graphique

4.6.1 Représentation des états

Un diagramme d'état possédant une structure de type « arbre », nous avons représenté l'esemble des états d'un diagramme à l'aide d'un `JTree`.

Pour cela, il faut parcourir à la création de la fenêtre graphique l'ensemble du système et peupler l'arbre.

Le principal piège de ce parcours est qu'il ne faut pas se contenter, pour un état, de parcourir chacun de ses suivants, car dans ce cas, si une transition forme une « boucle »,

le parcours se fera à l'infini ! Il faut donc adopter un parcours de graphe, en stockant les états qui ont déjà été parcourus.

5 Interface spécifique au diagramme d'état du clignotant

Cette interface spécifique a été construite de la même façon que l'exemple fait en TP (Jeu de Morpion), contrairement à la vue généraliste developpée sous l'application Netbeans.

Elle est similaire à l'interface généraliste , l'utilisateur est libre de choisir les évènements à stocker,toutefois la vue est un peu différente de la généraliste. L'ergonomie de l'application est donnée en figure 6

FIG. 6 – Ergonomie de l'application spécifique aux clignotants

L'exemple est relativement simple et ne presente pas d'indéterminisme. En effet, il y a indéterminisme que si à partir d'un même état actif un évènements peut produire plusieurs chemins différents vers des états différents, ce qui n'est pas le cas dans notre exemple.

5.1 Fenêtre principale

Les composants utilisés sont :

Jbutton : un élément qui peut contenir une image ;

Jframe : fenêtre principale d'une application Swing composé :
- d'un contenu (Container) accessible par get
- d'une barre de menus

Lors du lancement du programme, la fenêtre qui s'affiche permet à l'utilisateur d'interagir avec le diagramme.

L'utilisateur dipose de plusieurs boutons situés en bas de la fenêtre :

GAUCHE : Permet d'activer le feu gauche

DROITE : Permet d'activer le feu droit

WARNING : Permet d'activer les feu de détresse

SIMULER : Permet de faire avancer la simulation d'un pas de simulation

QUITTER : Permet de quitter le programme

5.2 Diagramme d'état

Ce diagramme d'état 7 décrit le comportement des feux clignotants gauches et droits d'un véhicule lorsque le conducteur actionne la manette des clignotants ainsi que les feux de détresse.

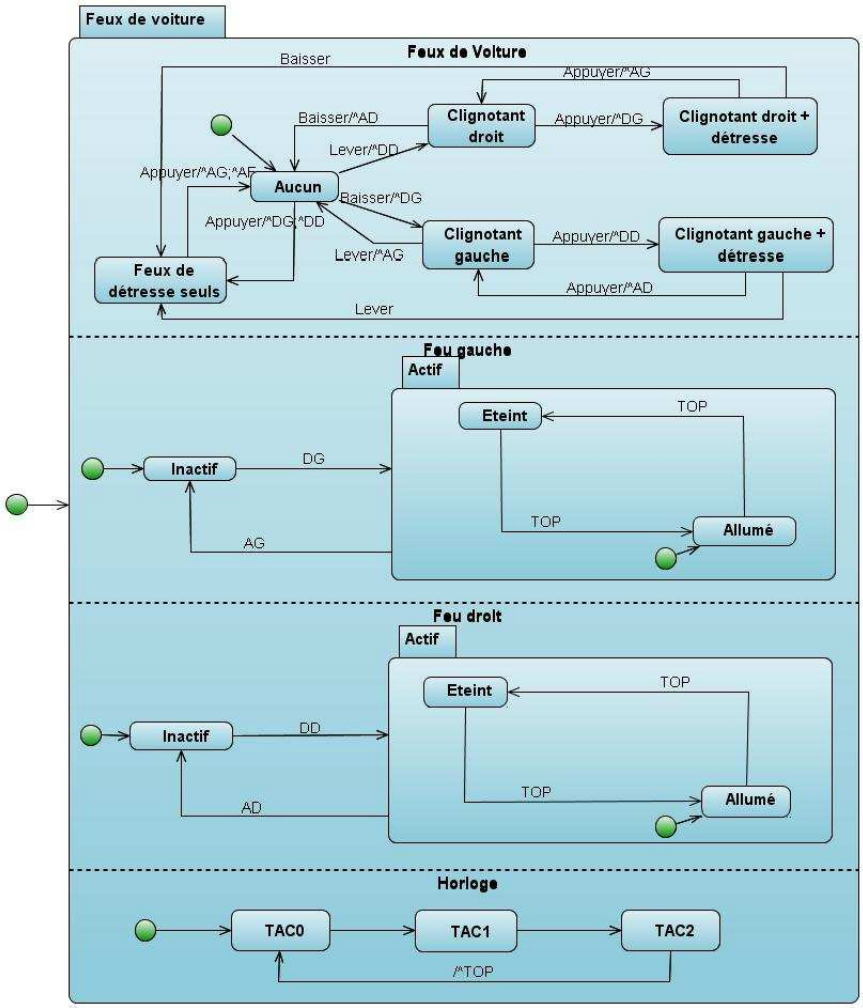

FIG. 7 – Diagramme d'état des clignotants

6 Manuel utilisateur

6.1 Création du diagramme

Pour pouvoir simuler un diagramme d'état, l'utilisateur doit écrire une classe dans laquelle il va construire son diagramme.

On explique ici comment créer un système de manière globalement ascendante en considération de notre diagramme de classe et des méthodes que nous avons définies. Il s'agit de créer d'abord tous les états du système, puis les super-états ou les diagrammes qui en sont composés. Comme les états sont composés d'actions et d'événements et que les

actions dépendent des événements et du système, on créera en tout premier lieu le système (dont on ne précise que le nom et que l'on remplira au fur et à mesure avec les nouveaux objets) et on crée les événements puis les actions, avant de créer les états.
Les étapes sont donc :

Créer le système :
Il faut utiliser le constructeur de la classe Systeme : `Systeme(String nomSysteme)`.

Créer les événements :
Un événement est caractérisé par son nom. On le construit à l'aide du constructeur de la classe `EvenementNom` : `new EvenementNom(String nomEvenement)`.

Créer les actions :
Nous ne traitons que les actions qui émettent des évenements. Une telle action est composée de l'événement `evt` qu'elle émet et du système `sys` auquel est ajouté cet événement lorsque l'action est exécutée. On construit donc une action de la manière suivante : `new Action(Evenement evt, Systeme sys)`.

Créer les états élémentaires :
Un état élémentaire est caractérisé par son nom et sa liste de transitions. Mais à cette étape, comme tous les états (ou super-états ou diagrammes ET) ne sont pas encore construits, on n'a pas encore crée les transitions. On construit un état comme suit : `new Etat(String nomEtat)`.

Créer les transitions entre les états élémentaires :
Pour créer une transition `transition` entre deux états élémentaires `e1` et `e2` (`e1` vers `e2`), on fait appel aux constructeurs de la classe Transition. Il y en a deux, selon que le franchissement de la transition se fait sous l'occurence d'un événement `evenement`, ou pas. Dans le premier cas, on écrit : `transition = new Transition(e2,evenement)` sinon `new Transition(e2)`. Si au franchissement de cette transition, on émet l'action `action` on a besoin de l'appel suivant : `transition.addAction(action)`. Enfin, pour ajouter cette transition à la liste des transitions de `e1`, on appelle : `e1.addTransition(transition)`.

Créer les états initiaux :
Pour créer un état initial vers un état `etat` (ou un diagramme), il faut d'abord créer la transition `transition` vers `etat` (cf au-dessus). Comme un état initial est un pseudo-état, il n'y a pas d'événement associé à sa transition. On n'utilisera donc toujours le constructeur de la classe Transition qui prend seulement un état suivant en paramètre. Ensuite on pourra appeler le constructeur de la classe EtatInitial : `new EtatInitial(String nomEtatInitial, Transition transition)`.

Créer les états englobants (super-états, diagrammes ET) :
Pour les construire il suffit de leur ajouter des états Initiaux : `diagramme.addEtatInitial(EtatInitial e)`. Si c'est un diagramme ET, il y aura autant d'appel à cette méthode que d'états abstraits qui composent le diagramme.
NB : un super-etat est du type Diagramme.

Etc... :
On peut ensuite ajouter des transitions entre diagrammes, créer des états initiaux vers eux pour les ajouter à d'autre diagrammes...

Voici un exemple simple de construction d'un système (figure 8).

```
1
    Systeme sys = new Systeme("Diagramme");
```

```
Diagramme diag = sys.getDiagramme();

Evenement evt = new EvenementNom("evt");
Action emettreEvt = new Action(evt,sys);

Etat e1 = new Etat("e1");
Etat e2 = new Etat("e2");

Transition t12 = new Transition(e2,evt);
t12.addAction(emettreEvt);
e1.addTransition(t12);
Transition t21 = new Transition(e1,evt);
t21.addAction(emettreEvt);
e2.addTransition(t21);

Transition ti = new Transition(e1);
ti.addAction(emettreEvt);
EtatInitial ei = new EtatInitial("ei",ti);

diag.addEtatInitial(ei);
```

Listing 1 – Création du diagramme

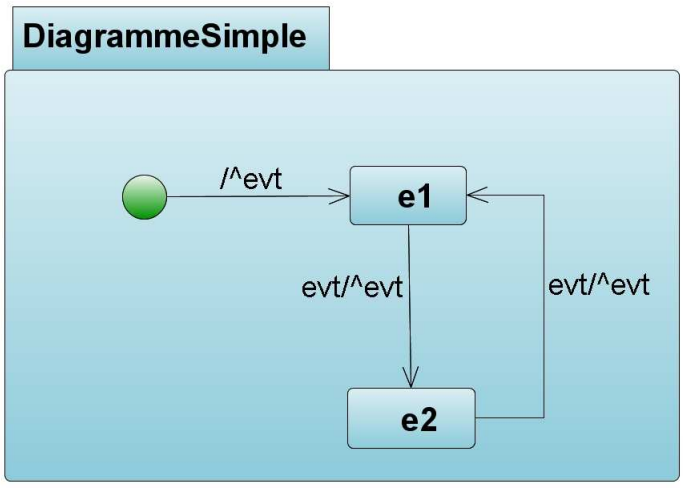

FIG. 8 – Exemple de construction d'un système

6.2 Ajout des vues

Pour visualiser les diverses actions se déroulant dans le diagramme, il faut lui ajouter des vues.

Les vues sont des instances des classes `Console`, ou `Graphique`. Leur constructeur prend en paramètre le système à observer.

Il faut ensuite « inscrire » ces vues au diagramme, via la méthode `inscrire` de la classe `Diagramme`.

Il est possible de définir une « vue principale », vue qui sera utilisée en cas d'indeterminisme. Pour cela, on utilisera la méthode `setObservateurPrincipal`, qui prend en paramètre un `Diagramme`, toujours de la classe `Diagramme`.

Pour finir, il faut initialiser le système, par appel à la méthode `initialiser`. Si cette étape n'est pas réalisée, alors le simulateur ne sera pas fonctionnel.

6.3 Interface Graphique Générale

6.3.1 Fenêtre Principale

Lorsque l'on lance le programme, la fenêtre qui s'affiche permet à l'utilisateur d'intéragir avec le diagramme.

Cette fenêtre est composées de deux onglets, qui géreront le diagramme et les évènements.

L'utilisateur dipose en plus de plusieurs boutons situés en bas de la fenêtre :

Continu Permet d'activer le mode de simulation continue

Pause Met en pause la simulation

Simuler Fait avancer la simulation d'un pas de simulation

RAZ Réinitialise le diagramme à son état initial

Quitter Permet de quitter le programme

6.3.2 Onglet Diagramme

Dans cette partie de l'interface, représentée en figure 9), il est possible de voir la liste des états du diagramme. Les états actifs sont marqués en police grasse, les inactifs en police normale.

La liste située au mileu de la fenêtre permet d'afficher l'ensemble des transitions qui ont été franchies pendant le pas de simulation.

Enfin, la liste située sur la droite affiche l'ensemble des évènements qui ont été générés pendant le pas de simulation précédent.

6.3.3 Onglet Evenements

Cette partie de l'interface (représentée figure 10) permet à l'utilisateur de définir des événements. Cela lui donnera ensuite la possibilité d'ajouter à la liste d'événements courants du système les événements de son choix parmi ceux qu'il aura définis. Il peut voir dans dans la liste en haut à gauche, les événements qui auront lieu au prochain pas de temps, ce qui lui permet de juger des événements utiles à ajouter. De toute façon, même s'il ajoute un événement qui va déjà avoir lieu, cet événement ne sera pas ajouté.

Dans le panel "Définir des événements", la combo box propose tous les événements susceptible de pouvoir agir sur le système. C'est pour le cas où l'utilisateur ne connaitrait pas

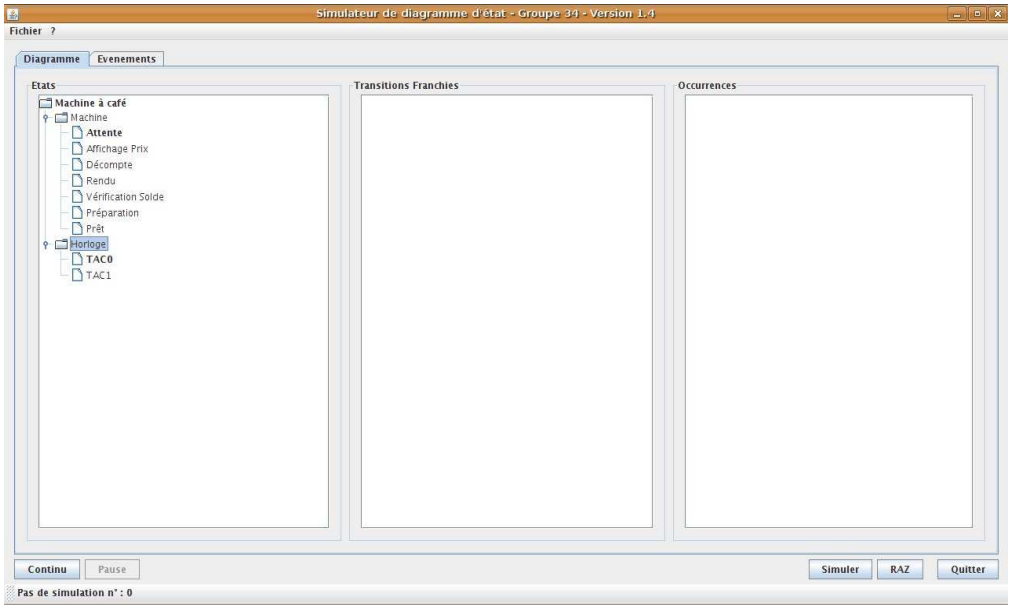

FIG. 9 – L'onglet de gestion du diagramme

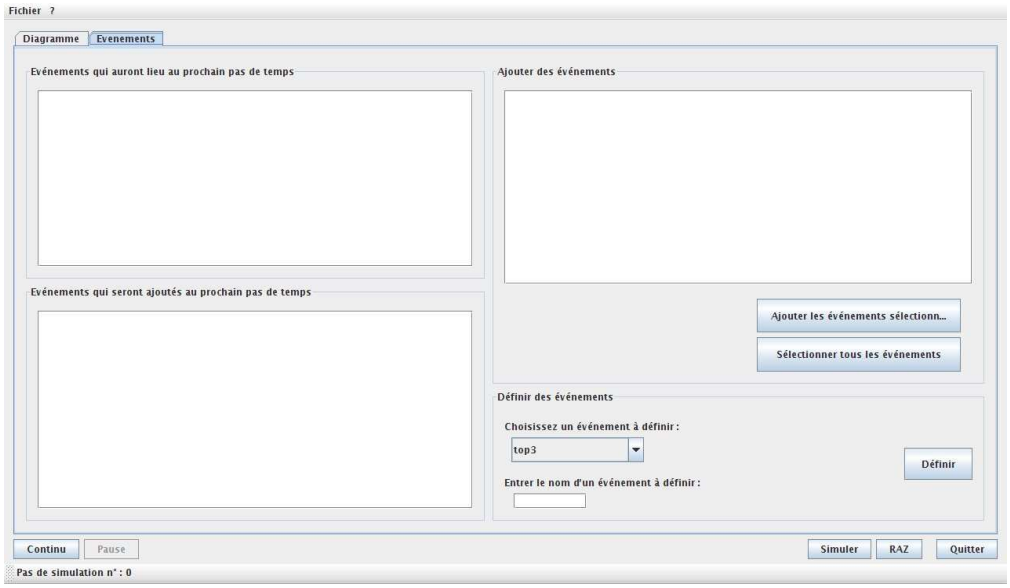

FIG. 10 – L'onglet de gestion des évènements

le nom des événements qui peuvent affecter le système. Il peut aussi définir un événement en rentrant son nom, ce qui lui permet de générer des événements qui n'auront pas d'effet sur le système (si le nom de l'événement qu'il définit ne correspond pas au nom d'un des événements qui sont visibles dans la combo box). L'appui sur le bouton **définir** privilégie la JtextField plutôt que la combo box. S'il y a du texte dans la zone de texte, ce n'est pas l'élément sélectionné de la combo box qui sera défini. On remarque qu'on ne peut pas définir deux fois le même événement.

7 Tests

7.1 Diagramme de test

7.1.1 Tester un diagramme simple

But : Tester la simulation sur un diagramme très simple qui comprend des événements et des actions afin de tester le bon fonctionnement de la majorité des classes du modèle. Le diagramme est représenté en figure 11.

Remarque : Voir DiagrammeSimpleTest dans le package diagramme.test.

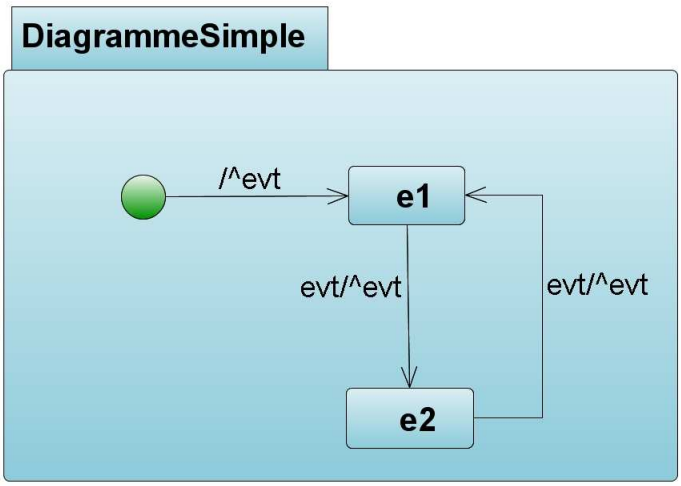

FIG. 11 – Test de diagramme simple

7.1.2 Tester un diagramme ET

But : On teste l'évolution simultanée des diagrammes, représenté en figure 12.

Remarque : voir DiagrammeTest dans le package diagramme.junit.

7.1.3 Tester un super-état

But : On teste la priorité dans le franchissement des transitions, représenté figure 13.

Remarque : voir DiagrammeTest dans le package diagramme.junit.

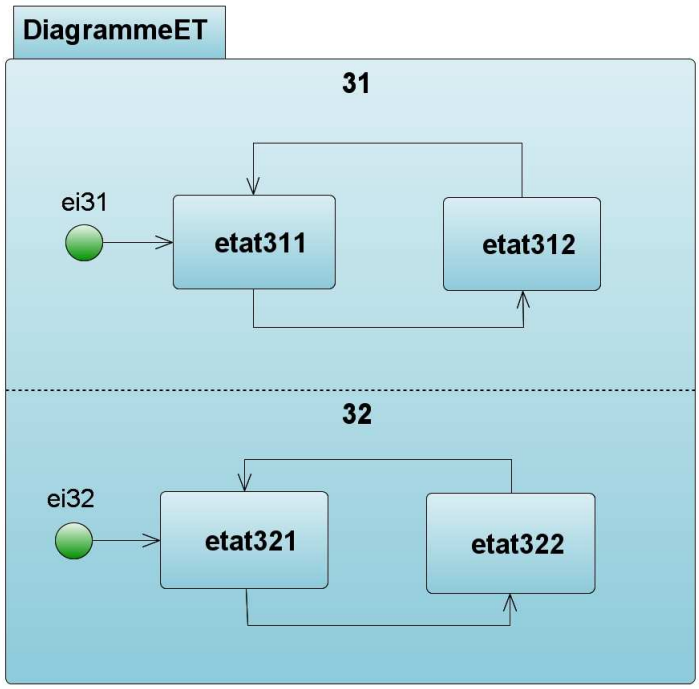

FIG. 12 – Test de diagramme ET

7.1.4 Tester l'indéterminisme

But : Tester la levée de l'indéterminisme, figure 14.

Remarque : voir EtatAbstraitTest dans le package diagramme.junit.

7.1.5 Tester l'initialisation

But : Tester que la simulation d'un diagramme inactif le laisse inchangé et tester l'initialisation d'un diagramme inactif, représenté en figure 15.

Remarque : voir DiagrammeTest dans le package diagramme.junit.

7.1.6 Tester la récupération des évènements

But : Tester la méthode de récupération de évènements, figure 16.

Remarque : voir DiagrammeTest dans le package diagramme.junit.

8 Tests

8.1 Diagramme de test

Pour tester notre simulateur, nous avons défini un diagramme de test, décrit sur la figure 17.

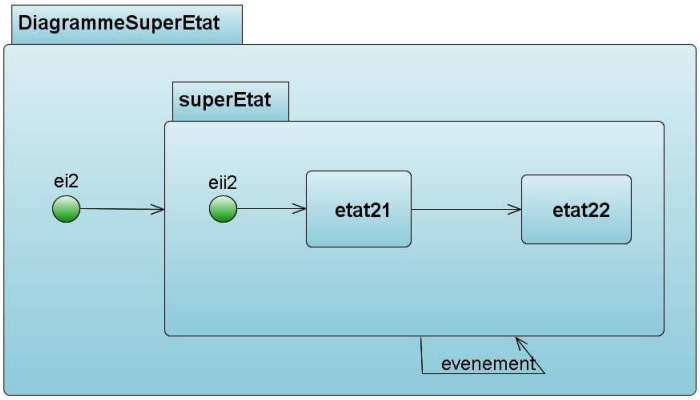

FIG. 13 – Test de diagramme Super Etat

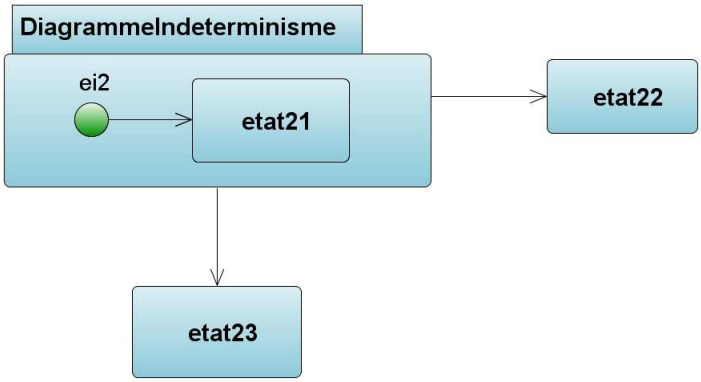

FIG. 14 – Test de diagramme Indeterminisme

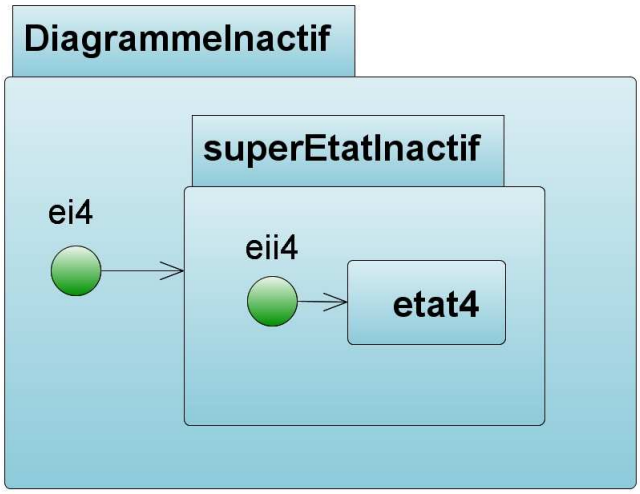

FIG. 15 – Test de diagramme Initialisation

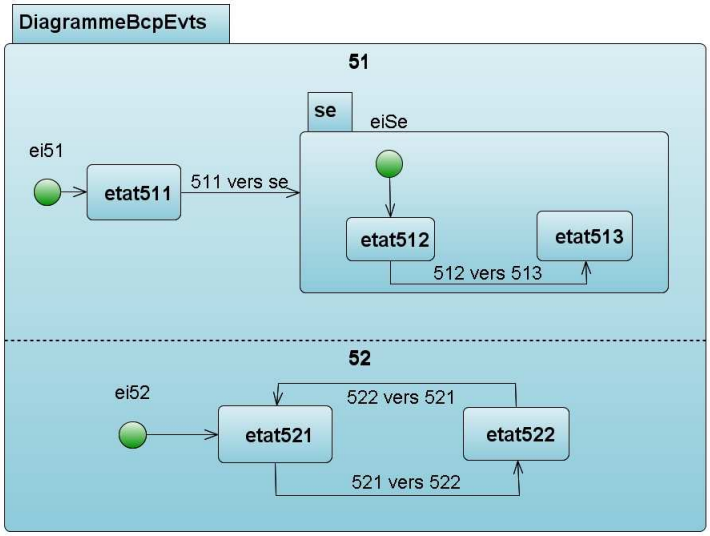

FIG. 16 – Test de récupération des évènements

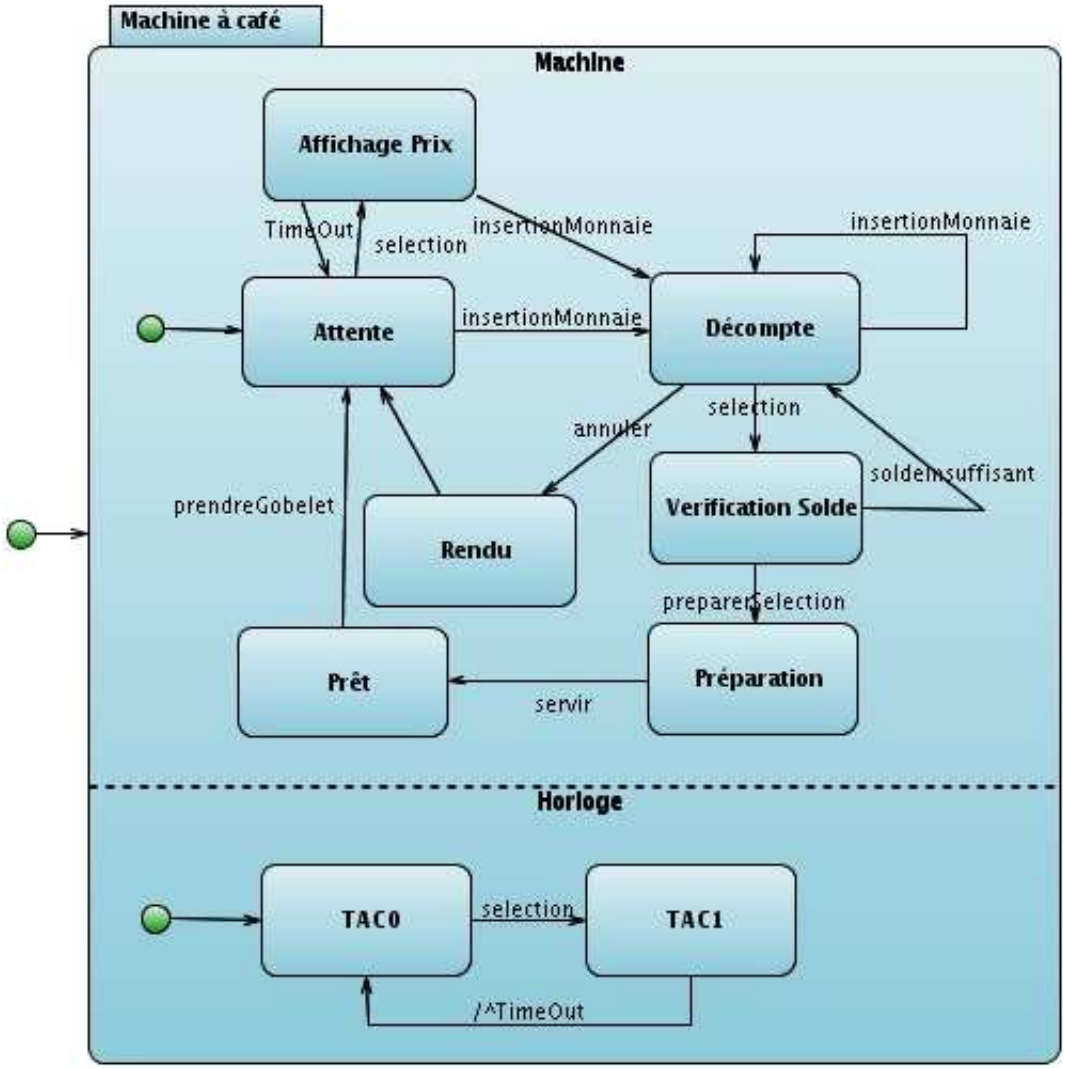

FIG. 17 – Diagramme d'état d'une machine à café

9 Conclusion

Ce projet a été l'occasion d'apprehender un premier travail en groupe, pas toujours facile.

C'est aussi une bonne façon d'approfondir notre vision de la modélisation UML, à la fois du point de vue de l'utilisation, car utilisée pour la phase d'analyse de ce projet, mais aussi vu le sujet de ce projet !

Enfin, le projet nous a permis de manipuler de façon plus concrète les divers outils vus au cours de l'année, et constitue une première approche de la création d'interfaces graphiques avec l'API `SWING`.

A Code sources

Ci-dessous se trouve le code de diverses classes du simulateur. Nous avons séléctionné les classes qui à notre avis étaient les plus importantes à la comprehension du code.

```
package diagramme.core;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;

import javax.swing.tree.DefaultMutableTreeNode;


/** Modélise un diagramme d'état UML.
 *
 * @author Lorraine
 * @version 1.2
 *
 */
public class Diagramme extends EtatAbstrait {

	/** Ensemble des états initiaux du diagramme. */
	private ArrayList<EtatInitial> etatsInitiaux;
	/* Permet de représenter les états ET */

	/** Ensemble des états actifs du diagramme. */
	private ArrayList<EtatAbstrait> etatsActifs;

	/** Construit un diagramme.
	 *
	 * @param n Le nom du diagramme
	 */
	public Diagramme(String n) {
		super(n);
		this.etatsActifs = new ArrayList<EtatAbstrait>();
		this.etatsInitiaux = new ArrayList<EtatInitial>();
	}

	/** Ajoute un état initial. */
	public void addEtatInitial(EtatInitial etat) {
		etatsInitiaux.add(etat);
	}

	/** Accesseur. */
	public ArrayList<EtatInitial> getEtatsInitiaux() {
		return etatsInitiaux;
	}

	public ArrayList<EtatAbstrait> getEtatsActifs() {
		return etatsActifs;
	}

	/** Fonction appelée à chaque pas de simulation.
	 *
	 * @param evenements La liste d'événements courants
	 */
	@SuppressWarnings("unchecked") // Evite le warning pour le cast utilisé a
vec "clone"
	public void simuler(Collection<Evenement> evenements) {
		Transition transitionFranchissable = null;
		// Shallow-copy pour éviter les java.util.ConcurrentModification
Exception
		Collection<EtatAbstrait> it = (ArrayList<EtatAbstrait>)etatsActi
fs.clone();
```



```
		for (EtatAbstrait o : it) {
			try {
				transitionFranchissable = o.getTransitionFranchi
ssable(evenements);
			} catch (IndeterminismeException e) {
				transitionFranchissable =
					observateurs.indeterminisme(e.getEtat(),
						e.getTransitions());
			}
			if (transitionFranchissable == null) {
				// pas de transition, ou  pas de transitions fra
nchissables, alors
				// on entre dans la structure si c'est un diagra
mme, ou on effectue
				// les actions si c'est  un état. o sera encore
actif au prochain pas
				// de temps dans tous les cas.
				o.simuler(evenements);
			}
			else {
				// On désactive l'état.
				o.desactiver();
				// On le supprime de la liste des états actifs.
				etatsActifs.remove(o);

				transitionFranchissable.franchir(etatsActifs);
				observateurs.transitionFranchie(o, transitionFra
nchissable);
			}
		}
	}

	/** Récupérer la liste de tous les événements pouvant affecter le
	 * diagramme. Utile pour PanelEvenements de l'interface graphique.
	 *
	 * @return la liste des événements pouvant affecter le diagramme.
	 */
	public Set<Evenement> recupererEvenements(Set<EtatAbstrait> visites) {
		HashSet<Evenement> listeEvenements = new HashSet<Evenement>();
		for (Transition t : this.getTransitions()) {
			if(!visites.contains(t.getSuivant())) {
				listeEvenements.add(t.getEvenement());
			}
		}
		for (EtatAbstrait e : etatsInitiaux) {
			listeEvenements.addAll(e.recupererEvenements(visites));
		}
		visites.add(this);
		return listeEvenements;
	}

	@Override
	public void activer() {
		super.activer();
		// On "nettoie" la liste des états actifs, au cas où...
		etatsActifs.clear();

		for (EtatInitial e : etatsInitiaux) {
			Transition transition = e.getTransitions().iterator().ne
xt();
			transition.franchir(etatsActifs);
		}
```

```
	}

	@Override
	public void desactiver() {
		super.desactiver();
		//
		for(EtatAbstrait e : etatsActifs) {
			// On désactive les états actifs
			e.desactiver();
		}
		etatsActifs.clear();
	}

	/** Méthode servant juste à mapper l'appel à initialiser depuis le Syste
me.
	 * * Initialise le diagramme par appel à activer()
	 * * Information des observateurs
	 */
	public void initialiser() {
		observateurs.initialiser();
		activer();
	}

	public void peupler(DefaultMutableTreeNode parent) {
		DefaultMutableTreeNode top = new DefaultMutableTreeNode(this);

		int taille = etatsInitiaux.size();
		DefaultMutableTreeNode diag = top;

		for(EtatInitial ei : etatsInitiaux) {
			// Si taille > 1, on mets les états initiaux dans des no
euds
			if(taille > 1) {
				diag = new DefaultMutableTreeNode(ei);
			}
			Set<EtatAbstrait> suivants = ei.getSuivants(new LinkedHa
shSet<EtatAbstrait>());
			suivants.remove(ei);
			for(EtatAbstrait e : suivants) {
				e.peupler(diag);
			}
			if(taille > 1) {
				top.add(diag);
			}
		}
		parent.add(top);
	}

	/** Méthode exécutée lorsqu'un pas de simulation a été effectué.
	 *
	 * @param tempsSimulation Le nombre de pas de simulation
	 */
	public void finPasSimulation(int tempsSimulation) {
		observateurs.finPasSimulation(tempsSimulation);
	}

	/** Méthode exécutée au début d'un pas de simulation. */
	public void debutPasSimulation() {
		observateurs.debutPasSimulation();
	}
}
```

```
package diagramme.core;

import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedHashSet;
import java.util.Set;

import javax.swing.tree.DefaultMutableTreeNode;

import diagramme.observateurs.GroupeObservateurs;
import diagramme.observateurs.Observateur;

/** Décrit un état abstrait.
 *
 * @author Audric Schiltknecht, Lorraine
 * @version 1.2
 *
 */
public abstract class EtatAbstrait {

	/** Le nom de l'état. */
	private String nom;

	/** Ensemble de transitions vers les états suivants. */
	private Collection<Transition> transitions;

	/** Ensemble d'actions à faire à l'entrée de l'évènement.*/
	private Collection<Action> onEntry;

	/** Ensemble d'actions à faire à la sortie de l'évènement.*/
	private Collection<Action> onExit;

	/** Sert lors du parcours du graphe. */
//	protected static Set<EtatAbstrait> visites;


	/** Constructeur utile pour la création ascendante du système.
	 *
	 * @param n nom de l'état ou du diagramme
	 */
	public EtatAbstrait(String n) {
		this.transitions = new ArrayList<Transition>();
		this.nom = n;
		this.onEntry = new ArrayList<Action>();
		this.onExit = new ArrayList<Action>();
		//visites = new HashSet<EtatAbstrait>();
	}

	/** Retourne la liste des actions onEntry à exécuter. */
	public Collection<Action> getOnEntry() {
		return this.onEntry;
	}

	/** Ajoute une action à effectuer à l'entrée de l'état. */
	public void addOnEntry(Action entree) {
		onEntry.add(entree);
	}

	/** Retourne la liste des actions OnExit à exécuter. */
	public Collection<Action> getOnExit() {
		return this.onExit;
	}
```



```
	/** Ajoute une action à effectuer à la sortie de l'état. */
	public void addOnExit(Action entree) {
		onExit.add(entree);
	}

	/** Retourne la liste des transitions. */
	public Collection<Transition> getTransitions() {
		return this.transitions;
	}

	/** Ajouter une transition à l'ensemble des transitions associées à l'état ou au
	 * diagramme.
	 *
	 * @param t la transition à ajouter.
	 */
	public void addTransition(Transition t) {
		transitions.add(t);
	}

	/** Lorsque l'état est activé.*/
	public void activer() {
		for(Action act : onEntry) {
			act.executer();
		}
		observateurs.activer(this);
	}

	/** Lorsque l'état est désactivé.*/
	public void desactiver() {
		for(Action act : onExit) {
			act.executer();
		}
		observateurs.desactiver(this);
	}

	/** Renvoie la liste des transitions franchissables.
	 * C'est l'ensemble des transitions dont l'événement
	 * déclencheur est dans la liste des événements courants.
	 * @param evenements la liste des événements courants.
	 * @return L'évenement que l'on peut franchir.
	 * @throws IndeterminismeException En cas d'indeterminisme.
	 */
	public Transition getTransitionFranchissable(Collection<Evenement> evenements) throws IndeterminismeException{

		ArrayList<Transition> transitionsFranchissables = new ArrayList<Transition>();

		for (Transition t : this.getTransitions()) {
			if (evenements.contains(t.getEvenement())) {
				transitionsFranchissables.add(t);
			}
		}

		if (transitionsFranchissables.size() > 1) {
			throw new IndeterminismeException(this, transitionsFranchissables);
		}

		Transition transitionFranchissable = null;
```

```
                // Si il n'y a aucune transition franchissable, on retourne null
                // Sinon, c'est la première qui est retournée
                if(transitionsFranchissables.size() != 0) {
                        transitionFranchissable = transitionsFranchissables.get(
0);
                }
                return transitionFranchissable;
        }

        /** Méthode retardée, qui sera appellée à chaque pas de simulation.
         *
         * @param evt Liste des évènements courants
         */
        public abstract void simuler(Collection<Evenement> evt);

        /** Méthode retardée, pour récupérer la liste des événements
         * susceptibles de modifier l'état.
         */
        protected abstract Collection<Evenement> recupererEvenements(Set<EtatAbs
trait> visites);

        /** Construit un noeud de l'arbre représentant le diagramme.
         *
         * @param parent Le parent duu noeud.
         */
        public abstract void peupler(DefaultMutableTreeNode parent);

        /** Retourne l'ensemble de suivants d'un état, ainsi que l'état lui-même
. */
        protected Set<EtatAbstrait> getSuivants(Set<EtatAbstrait> visites){
                LinkedHashSet<EtatAbstrait> suivants = new LinkedHashSet<EtatAbs
trait>();
                visites.add(this);
                suivants.add(this);
                for(Transition t : transitions) {
                        if(!visites.contains(t.getSuivant())) {
                                suivants.addAll(t.getSuivant().getSuivants(visit
es));
                        }
                }
                return suivants;
        }

        /** Fonction d'affichage. */
        public String toString() {
                return nom;
        }

//// Gestion des observateurs ////////////////////////////////////////////

        /** Groupe observant un état.
         * protected pour pouvoir appeller les méthodes de l'interface Observate
ur
         * depuis les classes dérivées de EtatAbstrait.
         * static pour que toutes les classes dérivant de EtatAbstrait possèdent
 les
         * possèdent ces observateurs.
         */
        protected static GroupeObservateurs observateurs
        = new GroupeObservateurs();
```



```
        /** Ajoute un observateur auprès du groupe. */
        public void inscrire(Observateur observateur) {
                observateurs.ajouter(observateur);
        }

        /** Retire un observateur auprès du groupe. */
        public void enlever(Observateur observateur) {
                observateurs.retirer(observateur);
        }

        /** Définit un observateur principal.
         * C'est celui qui sera utilisé en cas d'indeterminisme.
         * @param observateur L'observateur principal.
         */
        public void setObservateurPrincipal(Observateur observateur) {
                observateurs.setObservateurPrincipal(observateur);
        }

}
```

```
package diagramme.core;

import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;

import javax.swing.tree.DefaultMutableTreeNode;

/** Décrit un état du diagramme.
 *
 * @author Audric Schiltknecht, Lorraine
 * @version 1.3
 *
 */
public class Etat extends EtatAbstrait {

	/** Action a effectuer en fonction de la présence d'un évènement. */
	private HashMap<Evenement, Action> actions;
	/* Attention, dans ce cas, on ne peut pas avoir plusieurs actions executées
	 * à la même occurrence du même évènement.
	 */

	/** Construit un état à partir de son nom.
	 *
	 * @param nom Le nom de l'état
	*/
	public Etat(String nom) {
		super(nom);
		actions = new HashMap<Evenement, Action>();
	}

	/** Ajoute un couple (Evenement, Action) à la liste des actions
	 * internes de l'état
	 * @param evenement L'évènement déclanchant l'action
	 * @param action Action à executer sur l'occurrence de l'évènement
	 */
	public void addAction(Evenement evenement, Action action) {
		actions.put(evenement, action);
	}

	/** Cette méthode est appelée lorsque l'état est actif mais que sa liste de
	 * transitions ou bien sa liste de transitions franchissables est vide. Il faut alors
	 * gérer les actions associées à l'état. Attention, l'état reste actif !
	 *
	 * @param evt Ensemble des évènements courants.
	 *
	 */
	public void simuler(Collection<Evenement> evt) {
		// Effectuer l'action
		for (Evenement e : evt) {
			// On vérifie que la HashMap contienne bien la clé
			if(actions.containsKey(e)) {
				actions.get(e).executer();
			}
		}
	}

	@Override
```



```
	public void peupler(DefaultMutableTreeNode parent) {
		parent.add(new DefaultMutableTreeNode(this));

	}

	@Override
	public Set<Evenement> recupererEvenements(Set<EtatAbstrait> visites) {
		HashSet<Evenement> listeEvenements = new HashSet<Evenement>();
		visites.add(this);
		for (Transition t : this.getTransitions()) {
			listeEvenements.add(t.getEvenement());
			if(!visites.contains(t.getSuivant())) {
				listeEvenements.addAll(t.getSuivant().recupererEvenements(visites));
			}
		}
		return listeEvenements;
	}

}
```

```
package diagramme.core;

import java.util.Collection;
/** Exception levée en cas d'indeterminisme.
 *
 * @author Lorraine
 * @version 1.1
 *
 */
public class IndeterminismeException extends Exception {

        private static final long serialVersionUID = 1L;

        /** Etat source de l'indeterminsime. */
        private EtatAbstrait etat;

        /** Ensemble des transitions franchissables depuis cet état. */
        private Collection<Transition> transitions;

        /** Construit l'exception.
         *
         * @param etat Etat possedant plusieurs transitions franchissables.
         * @param t Ensemble des transitions franchissables.
         */
        public IndeterminismeException(EtatAbstrait etat, Collection<Transition>
 t) {
                super("L'état " + etat + " possède plusieurs transitions franchissables !");
                this.etat = etat;
                this.transitions = t;
        }

        /** Retourne l'ensemble des transitions. */
        public Collection<Transition> getTransitions() {
                return this.transitions;
        }

        /** Retourne l'état. */
        public EtatAbstrait getEtat() {
                return this.etat;
        }

}
```

```
package diagramme.observateurs;

import java.util.Collection;

import diagramme.core.EtatAbstrait;
import diagramme.core.Transition;


/** Interface d'un observateur.
 *
 * @author Audric Schiltknecht
 * @version 1.0
 *
 */
public interface Observateur {

        /** Indique que le pas de simulation commence. */
        public void debutPasSimulation();

        /** Indique que le pas de simulation est terminée.
         *
         * @param tempsSimulation Nombre de pas de simulation écoulés.
         */
        public void finPasSimulation(int tempsSimulation);

        /** Indique que l'état a été activé. */
        public void activer(EtatAbstrait etat);

        /** Indique que l'état a été desactivé. */
        public void desactiver(EtatAbstrait etat);

        /** Initialise ou RAZ du système. */
        public void initialiser();

        /** Indique que la transition a été franchie.
         *
         * @param etatSource L'état source de la transition.
         * @param t La transition franchie
         */
        public void transitionFranchie(EtatAbstrait etatSource, Transition t);

        /** Méthode appelée en cas d'indéterminisme.
         *
         * @param etat L'état courant
         * @param transitions L'ensemble des transitions franchissables
         */
        public Transition indeterminisme(EtatAbstrait etat, Collection<Transitio
n> transitions);
}
```

```
package diagramme.core;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

/** Représente le système.
 * Diagramme d'état + ensemble d'évènements.
 * @author Audric Schiltknecht, Lorraine
 * @version 1.3
 *
 */
public class Systeme {

	/** Diagramme modélisé. */
	private Diagramme diagramme;

	/** Ensemble d'évènements courants. */
	private Collection<Evenement> evenements;

	/** Ensemble des évènements suivants. */
	private Collection<Evenement> evenementsSuivants;

	/** Nombre de pas de simulation écoulés. */
	private int pasSimulation;


	/** Pour la construction ascendante. Juste après avoir créé les événemen
ts, on crée
	 * le système avec un diagramme vide, car on a besoin du système (de sa
liste
	 * d'événements) pour créer les actions.
	 *
	 * @param nom Le nom du diagramme
	 */
	public Systeme(String nom) {
		this.diagramme = new Diagramme(nom);
		this.evenements = new ArrayList<Evenement>();
		this.evenementsSuivants = new ArrayList<Evenement>();
		pasSimulation = 1;
	}

	/** Pour la construction ascendante. Après avoir construit tout le diagr
amme principal,
	 * on peut l'affecter au champ diagramme du système préalablement constr
uit.
	 *
	 */
	public Diagramme getDiagramme(){
		return this.diagramme;
	}

	/** Ajouter un événement à la liste d'événements suivants du système.
	 *
	 * @param e l'événement à ajouter.
	 */
	public void addEvenement(Evenement e) {
		evenementsSuivants.add(e);
	}

	/** Retourne la liste des évènements. */
```



```
	public Collection<Evenement> getEvenements() {
		return this.evenementsSuivants;
	}

	/** Méthode appelée à chaque pas de simulation. */
	public void simuler() {

		// On avertit les observateurs
		diagramme.debutPasSimulation();

		// On échange les évènements.
		evenements = evenementsSuivants;
		evenementsSuivants = new ArrayList<Evenement>();

		// Ne pas oublier d'ajouter l'evenement NOP !
		evenements.add(EvenementNOP.createEvenementNOP());

		// On lance la simulation
		diagramme.simuler(evenements);
		// On avertit les observateurs
		diagramme.finPasSimulation(pasSimulation++);

	}

	/** Parcourir tout le système afin de récupérer la liste
	 * de tous les événements pouvant l'affecter. Utile
	 * pour PanelEvenements de l'interface graphique.
	 *
	 * @return la liste des événements pouvant affecter le système.
	 */
	public Set<Evenement> recupererEvenements() {
		return diagramme.recupererEvenements(new HashSet<EtatAbstrait>()
);
	}

	/** Sert à initialiser le système.
	 * <p>Cette méthode doit être appelée une fois avant
	 * l'appel à la méthode simuler, sinon il ne se
	 * passera rien.</p>
	 */
	public void initialiser() {
		diagramme.initialiser();
		pasSimulation = 1;
	}

}
```

```
package diagramme.core;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Vector;

/** Décrit une transition entre deux états.
 *
 * @author Audric Schiltknecht, Lorraine
 * @version 1.0
 *
 */
public class Transition {

	/** L'état suivant cette transition. */
	private EtatAbstrait suivant;

	/** Évènements autorisant le franchissement de la transition. */
	private Evenement evenement;

	/** Ensemble des actions à effectuer lors du franchissement. */
	private Collection<Action> actions;

	/** Construit une transition à partir de son état suivant. */
	public Transition(EtatAbstrait etatSuivant) {
		this.suivant = etatSuivant;
		this.evenement = EvenementNOP.createEvenementNOP();
		this.actions = new Vector<Action>();
	}

	/** Construit une transition vers un état suivant, avec un évènement. */
	public Transition(EtatAbstrait etat, Evenement evenement) {
		this(etat);
		this.evenement = evenement;
	}

	/** Ajoute une action à la transition.
	 *
	 * @param a Action à ajouter
	 */
	public void addAction(Action a) {
		actions.add(a);
	}

	/** Retourne l'état cible. */
	public EtatAbstrait getSuivant() {
		return suivant;
	}

	/** Retourne l'évènement associé à la transition. */
	public Evenement getEvenement() {
		return this.evenement;
	}

	/** Franchir la transition.
	 * Effectue les opérations lors de l'activation d'un état :
	 *		* Ajout de l'état à la liste des états actifs du diagram
me
	 *		* Activation de l'état
	 * @param etatsActifs Liste des états actifs à laquelle on ajoute l'état
 suivant
	 */
```



```
	public void franchir(ArrayList<EtatAbstrait> etatsActifs) {
		EtatAbstrait suivant = getSuivant();

		// Effectuer les actions
		for (Action a : this.actions) {
			a.executer();
		}

		//Execution des actions onEntry de l'état
		suivant.activer();

		// Ajouter l'état suivant à la liste d'états actifs
		etatsActifs.add(suivant);
	}

	/** Fonction d'affichage. */
	public String toString() {

		String retour = "--(" + evenement.toString();

		if(!actions.isEmpty()) {
			retour += "/";
			for(Action a : actions) {
				retour +="^" + a.toString() + ";";
			}
			// On retire le dernier ';'
			retour = retour.substring(0, retour.length()-1);
		}


		retour += ")--> " + suivant.toString();

		return retour;
	}

}
```