TRAVAUX DIRIGES DE SIMULATION ENSTA BRETAGNE - PARTIE 2 ANNEE 2019-2020

Enseignant : Olivier VERRON (simuenstabretagne@gmail.com)

1. PREAMBULE

Cette partie du TD va vous conduire à apprendre à vous servir de 3 bibliothèques fournies et à utiliser dans le cadre des parties 1 et 2.

2. OBJET DE L'EXERCICE

- Utiliser un moteur de simulation événementiel pur plus complexe que le TD précédent
- Créer une application de simulation simple avec ce nouveau framework

3. PRE REQUIS

Le seul prérequis qui sera exposé ici est celui de la notion d'Interface fonctionnelle.
Il s'agit d'interface Java ayant une seule fonction et annotée avec une annotation FunctionalInterface

```
@FunctionalInterface
public interface NotifySimEvent {
	void notifySimEvent(ISimEvent ev);
}
```

Une telle interface peut s'utiliser comme les autres.
Mais elle a une propriété syntaxique d'usage très intéressante.

Ainsi une classe implémentant une méthode correspondant exactement à la signature de cette unique méthode peut être utilisée comme un pointeur de fonction. On a alors un code plus élégant qu'en utilisant des classes anonymes.

Ci-dessous un exemple d'utilisation (en rouge gras):

```
class Test {
	void fonctionLevantEvenement() {
		unevenementmemorise = new SimEvent() ;
		fire(Test::mePrevenirDe) ;
	}
	SimEvent unevenementmemorise ;
	void fire(NotifySimEvent mamethode) {
		Ev.notifySimEvent(unevenementmemorise) ;
	}

	void mePrevenirDe(ISimEvent ev) {
		System.out.println(« Je vois un événement »+ev.toString()) ;
	}
```

```
}
```

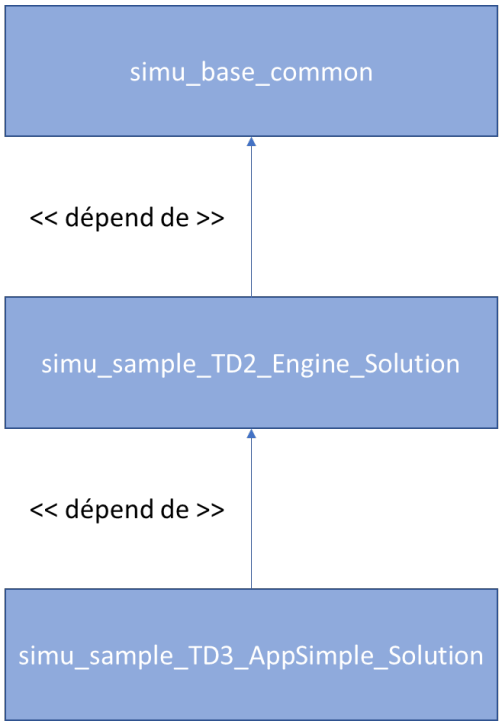
Le corrigé utilise cette notation à plusieurs reprises.

4. PREPARATION DE L'ESPACE DE TRAVAIL

Contrairement au TD1, cette fois on sépare le framework et l'application par deux projets différents.

On importe le projet `Simu_sample_TD2_Engine.zip` situé sur moodle.

On créée un nouveau projet `Simu sample_TD3_AppSimple` en rajoutant la dépendance vers `simu_sample_TD2_Engine_Solution`.



Vérifiez comment est fait le build path de Engine Solution via le menu Configure BuildPath > Projects pour créer les dépendances, puis par Order and Export pour réexporter le projet que vous souhaitez (point vue en TD0). Vous verrez que `simu_base_common` est réexporté.

4.1 Simu_sample_TD2_EngineSimple

L'objet de cette partie est de vous faire observer la structure du code.

Le framework est contitué d'un ensemble de plusieurs composantes :

- Un moniteur abstrait
- Un moteur de simulation
- Un concept abstrait d'entité de simulation
- Un scénario
- Un évenement

4.2 Simu_sample_TD2_AppSimple

Ce projet a pour but d'héberger l'application test simple.

Elle hébergera une spécialisation :

- Du moniteur
- D'entité
- De scénario
- D'événement

Il s'agira de réaliser un Helloworld.

5. PROJET DU MOTEUR DE SIMULATION SIMPLE

Il s'agira de retrouver les fonctions qui vont être décrites ci-dessous dans le code fournit dans `simu_sample_TD2_Engine_Solution.`

5.1 Exigence générales

Les exigences ci-dessous expriment les grands services attendus du framework de simulation :

Ex 1. Le moteur de simulation est instancié par un moniteur

Ex 2. Le moniteur invoque l'avancée du temps logique du moteur de simulation selon une boucle de contrôle « As Fast As Possible ».

Ex 3. Le moteur de simulation initialise au moment de son instanciation un germe de nombre aléatoire utilisé par la bibliothèque de génération de nombres aléatoires fournie en données d'entrée.

Ex 4. Le moteur de simulation identifie le temps de la simulation par un temps logique.

Ex 5. Le moteur de simulation incrémente le temps de la simulation selon les événements datés qui se produisent.

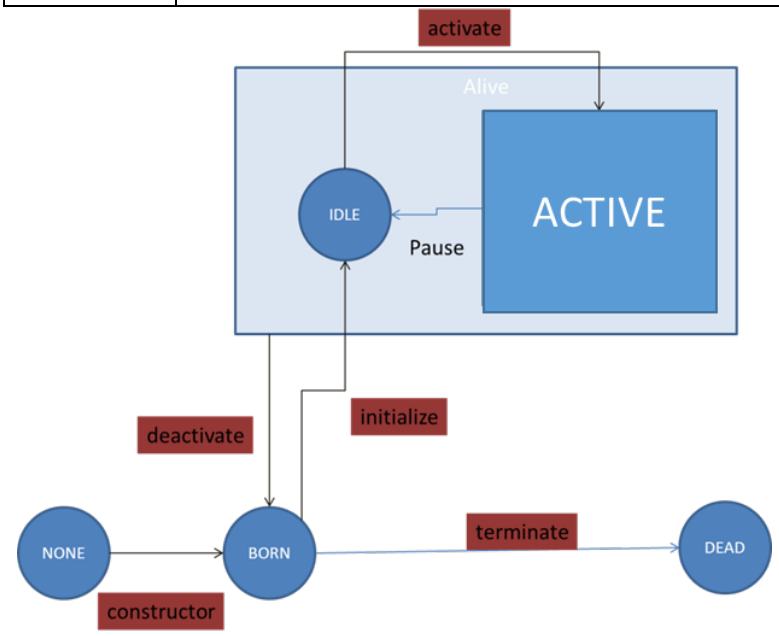
Ex 6. Le moteur de simulation anime des entités de simulation. Les entités de simulation peuvent avoir des sous entités aux cycles de vie imbriqués dans leur parente. Ainsi une entité parente ne peut être désactivée avec un fils actif.

Ex 7. Chaque entité suit le cycle de vie ci-dessous :

Etat	Description
NONE	L'entité en cours de constitution (constructeur).
BORN	L'entité est créée mais ne peut poster d'événements.
IDLE	L'entité est initialisée et peut poster des événements
ACTIVE	L'entité est en cours de traitement des événements qui la concerne.
DEAD	L'entité n'a plus de raisons d'exister. Elle n'est plus connue du moteur de simulation.

Transition	Description

Constructor	Invocation du constructeur par le programme principal. Les paramètres du constructeur représentent les données techniques structurantes de l'entité.
Initialize	Invocation par le moteur. Initialisation de l'entité. Les paramètres de l'initialisation représentent des données ne structurant pas l'entité.
Activate	Invocation par le moteur. A l'activation, les événements de l'entité peuvent être postés à partir de cet événement.
Pause	Invocation par le moteur. La Pause peut être invoquée suite à demande d'utilisateur (optionnel) via le moniteur.
Deactivate	Invocation par le moteur. Désactive en cascade les autres entités dont elle est parente. Retire les événements ultérieurs à l'événement présent de l'échéancier.
Terminate	Invocation par le moteur. L'entité enlève toute référence à d'autres. Ceci permet au garbage collector de vider la mémoire. L'entité ne doit plus être référencée par aucun objet.



Ex 8. Chaque entité de simulation notifie au moteur de simulation l'ensemble de ses événements explicites temporels.

Ex 9. Le moteur de simulation active l'entité de simulation dans l'ordre d'occurrence des événements qui la concernent : il garantit l'application du principe de causalité.

Ex 10. Le moteur réalise une journalisation des événements en s'appuyant sur la bibliothèque de log fournie en données d'entrée.

Ex 11. Le moteur permet de trouver d'autres entités de simulation par requêtage.

Ex 12. Le moteur de simulation désactive l'ensemble des entités lorsque la simulation ne comprend plus aucun événement temporel à exécuter.

Ex 13. Le moteur de simulation désactive les entités à une date logique maximale.

Ex 14. Le moteur de simulation détruit de la mémoire l'ensemble des entités de simulations lorsque l'ensemble des entités de simulation sont désactivées.

5.2 Moniteur

Sa principale fonction est pouvoir exécuter successivement une multitude de scénario.
Il porte le `main()` du programme de simulation.
C'est lui qui va instancier :

- le scénario à jouer.
- le moteur de simulation

Ses principales fonctions seront donc :

- `Void run(List<SimScenario>, long)`
- `Void run(SimScenario, long)`
- `Void terminate(boolean)`

La fonction run veillera à lancer l'`init()` puis le `simulate()` du moteur.

5.3 Moteur de simulation

Il implémente l'interface `ISimEngine` suivante :

Cette fonction va initialiser la simulation, provoquer l'instanciation des différentes entités puis leur initialisation.

```
void init(IScenario currentScenario,SimScenarioInit initScenario);
```

Après initialisation il peut renvoyer l'identifiant du scénario qu'on lui a fournit.

```
ScenarioId getScenarioId();
```

Après l'initialisation il s'agit d'activer la simulation en propageant l'activation à toutes les entités de la simulation

```
void activateSimulation();
```

La fonction ci-dessous est le cœur du moteur. Elle va boucler sur les événements postés par l'activation. Pour rappel du cours, c'est par l'activation provoquée de chaque entité que événements initiaux vont être créés. Et ce sont ces événements qui produisent une réaction en chaine jusqu'à épuisement des événements ou l'atteinte de la fin de la durée demandée de la simulation.

```
void simulate();
```

La fonction qui renvoie la date de simulation. Le moteur est le garant du temps logique en cours.

```
LogicalDateTime SimulationDate();
```

La fonction s'assure que les entités soient désactivées proprement.

```
void deactivateSimulation();
```

La fonction qui s'assure que tout s'arrête proprement, que les entités passent dans les bons états. Le boolean va configurer les entités pour un redémarrage éventuel.

```
void terminate(boolean restart);
```

La fonction de requêtage qui permet de trouver une entité. La requête répond à une interface fonctionnelle de type `boolean filter(ISimObject o);`

```
List<ISimObject> requestSimObject(SimObjectRequest openedSalonRequest);
```

Les deux fonctions ci-dessous permettent de s'abonner à l'événement de création d'une entité. L'objet renvoyé est une entité de simulation à l'état BORN. L'entité ne pourra être vraiment utilisée qu'après activation.

```
void AddSimObjectAddedListener(SimObjectAddedListener listener);
void RemoveSimObjectAddedListener(SimObjectAddedListener listener);
```

La fonction renvoie le générateur de nombres aléatoires

```
MoreRandom getRandomGenerator();
```

5.4 Entité de simulation

Date courante issue du moteur et identifiants

```
LogicalDateTime CurrentDate();
int getSimObjID();
String getName();
String getFullName();
```

Cœur de l'entité ces méthodes permettent la transmission des événements au moteur.

```
void Post (ISimEvent ev);
void Post (ISimEvent ev,LogicalDateTime t);
void Post (ISimEvent ev,LogicalDuration dt);
void UnPost(ISimEvent ev);
void UnPostAllEvents();
void terminate(boolean restart);
```

Relations de filiation

```
List<IEntity> getChildren() ;
IEntity getParent() ;
```

Service d'activation utilisé par le moteur de simulation ou une entité parente

```
void activate();
void deactivate();
```

Ensemble de listes d'abonnements pour lesquelles il suffit de placer une fonction répondant à l'interface fonctionnelle de la liste. Ces événements sont levés en fonction du cycle de fonctionnement de l'entité.

```
List<CreationNotification> OnCreating();
List<CreationNotification> OnCreated();

List<InitializationNotification> OnInitializing();
List<InitializationNotification> OnInitialized()
List<ActivationNotification> OnActivating();
List<ActivationNotification> OnActivated();

List<TerminatingNotification> OnTerminating();
List<DeactivationNotification> OnDeactivating();

List<DeactivationNotification> OnDeactivated();
boolean IsAlive();
boolean IsActive ();
```

Générateur de nombres aléatoires

```
MoreRandom RandomGenerator();
```

5.5 Evénement

Un événement est lié à son Entité productrice

```
IEntity Owner();
```

Un événement a une date d'occurence

```
LogicalDateTime ScheduleDate();
```

Un événement une action sur son entité productrice (owner)

```
void Process();
```

Un événement est initialisé par la Simentity productrice

```
void initialize(IEntity simObject, LogicalDateTime t);
```

Un événement est détruit après exécution à moins d'être reposté.

```
void terminate();
```

Un événement peut être reprogrammé

```
void resetProcessDate(LogicalDateTime simulationDate);
```

5.6 Scénario

Un scénario est caractérisé par les services suivants leurs noms se suffisants à eux-mêmes:

```
String getName();
long getSeed();
ScenarioId getScenarioId();

LogicalDateTime getStartDateTime();
LogicalDateTime getEndDateTime();

void terminate(boolean restart);
```

6. PROJET D'APPLICATION DE SIMULATION

Il vous est demandé de simuler un groupe d'étudiant le jour de la rentrée, ils ne se connaissent pas.
Les étudiants arrivent au fur et à mesure dans la journée et se découvrent mutuellement au fil de l'eau.

Les étudiants ont des points d'intérêt : les films préférés.
Si les films qu'ils préfèrent ne sont pas connus de l'autre, alors ils ne se parleront pas.
Si les films qu'ils préfèrent sont connus de l'autre alors ils se parlent de manière aléatoire selon un écart situé entre 2 et 6 secondes.

A l'initialisation on décide quels films sont préférés pour chaque personne.

Vous êtes invités à regarder une manière de réaliser cette simulation en ouvrant le corrigé `simu_sample_TD3_AppSimple_Solution.`

Noter par exemple comment on peut s'abonner à la création d'un objet,

```
getEngine().AddSimObjectAddedListener(this::attendreNouvelArrivant);
```

puis à son activation dans l'activation de l'étudiant.

```
s.OnActivated().add(this::accueillirNouvelArrivantActif);
```