Université Paris Sud - Développement Logiciel - L2
TP4: Les threads

Très important : Ce TP est noté. Vous devrez envoyer par e-mail les sources du TP dans une archive à **oleg.lodygensky@lal.in2p3.fr** avant le **dimanche 17 février 2013 à 23h59**. Il s'agit d'un travail personnel à effectuer en monôme ; les binômes sont donc interdits.

Résultats attendus

Ce TP fournit le fichier `build.xml`, utilisable avec `ant`, permettant de compiler, d'extraire la *JavaDoc* des fichiers sources *Java*.

Il est impératif que vos travaux et leurs documentations soient compilables avec cet outil. Pour cela, il faut et il suffit de ne pas toucher au fichier `build.xml`. Si vous le modifiez, il est de votre responsabilité que les **cibles ant** décrites dans le préambule fonctionnent correctement.

Préambule

Référence le logiciel *Nibbles*, dont ce TP est inspiré, est disponible ici :
`http://zetcode.com/tutorials/javagamestutorial/snake.`

Les codes sources sont disponibles à l'adresse suivante :
http ://www.xtremweb-hep.org/lal/TP4.zip
Téléchargez et décompressez cette archive

Les cibles ant L'archive du TP contient le fichier *build.xml*, dans le répertoire *Snake*, utilisable avec l'outil `"ant"`. Ce fichier contient les cibles suivantes :

- build : pour compiler le projet
- clean : pour effacer les binaires du projet
- Nibbles : pour lancer l'exécution du projet
- sonar : pour vérifier la qualité du code logiciel
- doc : pour extraire la documentation Java des sources du projet

Par exemple, pour compiler le projet, tapez : **ant build**, dans le répertoire *Snake*.

Lisibilité du code produit : votre code doit être propre et lisible. Pensez à bien indenter votre code (Eclipse se charge de le faire pour vous grâce au raccourci clavier Ctrl+i) ; à utiliser des noms de package, d'interface, de classes, de méthodes et d'attributs qui soient lisibles et parlant ; à documenter votre code grâce à *JavaDoc*. Des points supplémentaires sont attribués aux codes clairs respectant ces conditions.

Documentation du code produit : votre code (interfaces, classes, méthodes, attributs etc.) doit être documenté avec la syntaxe *JavaDoc*. Un code non documenté sera pénalisé. Veuillez noter que l'outil *Sonar* considère comme une erreur de ne pas documenter une interface, classe, méthode ou attribut déclaré **public**.

Pour extraire la JavaDoc du projet, tapez : **ant doc**.

De la bonne utilisation de la console : votre code ne doit contenir aucun appel à `System.out.println`. L'utiliser vous retirera des points. Si vous avez besoin d'écrire sur la console, vous utiliserez la classe `Logger` de la librairie Java. A titre d'exemple, le code fourni utilise un attribut de ce type dans certaines classes. Il sera de votre responsabilité de développeur de faire attention à l'utilisation des niveaux de *log*.

L'archive du TP contient un fichier *log.config* que vous pouvez modifier en fonction de vos besoins de *logging*.

Installer et démarrer votre server sonar

L'archive de l'outil *Sonar* est disponible à l'adresse *http ://www.sonarsource.org*.

Téléchargez l'archive de *Sonar* et décompressez la. Le répertoire *bin* contient un ensemble de sous répertoires où vous trouverez les différents binaires pour lancer le serveur en fonction de votre plate-forme. Par exemple si vous installez Sonar sur une machine linux avec un processeur *Intel* 32 bits, vous lancerez :

```
bin/linux-x86-32/sonar.sh start
```

Audit de votre code source Après avoir lancé votre serveur *Sonar*, vous pouvez auditer la qualité du code logiciel du projet. Le fichier de compilation *build.xml* fourni dans l'archive du TP permet de générer les rapports d'audit. Pour lancer l'audit, tapez :

```
$> cd Snake
$> ant sonar
```

Vous pouvez ensuite accéder aux statistiques sur la page de votre serveur à l'adresse :

```
http://localhost:9000
```

L'audit de **votre** code logiciel à la fin de ce TP devra faire apparaitre :
- Rules compliance : supérieur à 70% ;
- Blocker violation : 0 ;
- Major violation : 0 ;
- Documentation *JavaDoc* : supérieur à 70%.

Introduction

On se propose d'améliorer le jeu *Nibbles* cité en référence, écrit en langage Java, dans lequel un joueur déplace un serpent dans une arène pour manger le plus de pommes possible.

Règle du jeu Le joueur dirige un serpent composé d'une tête et de plusieurs articulations ; il le contrôle avec les flèches *haut*, *bas*, *droite* et *gauche* du clavier. Le jeu démarre avec, sur l'arène, un serpent composé de trois articulations et une pomme. A chaque pomme mangée, le serpent grandit d'une articulation. Le serpent n'a pas le droit de se mordre la queue ; il ne doit pas non plus toucher un des bords de l'arène. Le jeu est fini dès que le serpent se mord la queue ou touche un bord de l'arène.

Règle du jeu "multi joueurs" Ces règles reprennent les précédentes et y ajoutent de nouvelles fonctionnalités. Le joueur n'est plus seul à déplacer un serpent dans l'arène. Nous allons y introduit un serpent "robot" contrôlé par l'ordinateur dont les déplacements sont aléatoires. En plus des règles de base, le joueur n'a pas le droit de toucher ce serpent robot. Le jeu est fini en cas de violation des règles de base, ainsi qu'en cas de collision avec le robot. Le robot peut manger les pommes ; si le robot viole une des règles, le jeu s'arrête.

Interface homme machine Vous n'avez pas à travailler sur l'interface graphique (*GUI*). Nous vous fournissons tous les fichiers sources et vous n'avez pas à modifier la GUI.

La version de base

Nous vous fournissons les sources du jeu de base dans cinq fichiers *Java* ; ils sont entièrement documentés en *JavaDoc* :

- `Nibbles` est la classe principale. Elle instancie l'arène et y insère les serpents.
- `Board` implémente l'arène et sa GUI. Cette classe gère le clavier pour contrôler le serpent du joueur. Vous n'aurez pas besoin de modifier la gestion du clavier.
- `Apple` définit la pomme. Nous y trouvons les méthodes :
  1. `getX()` et `getY()` pour obtenir ses coordonnées ;
  2. et `draw()`qui affiche la pomme sur la GUI.
- `Snake` est la classe abstraite définissant la notion de serpent. Nous y trouvons :
  - L'énumération *Direction* définissant les mouvements possibles du serpent.
  - Les méthodes
    1. `left(), right(), up()` et `down()` qui déplacent le serpent ;
    2. `checkCollision()` qui détermine si le serpent se mord lui même ou rencontre un bord de l'arène ;
    3. `eatApple()` détermine si le serpent mange la pomme, auquel cas le serpent gagne une articulation et une nouvelle pomme sera installée sur l'arène ;
    4. enfin, `draw()`qui affiche le serpent sur la GUI.

– `SnakeKeyboard` qui définit le serpent contrôlé par le clavier. Cette classe hérite de la classe *Snake* sans y ajouter aucune fonctionnalité. Elle permet d'instancier un serpent.

Lancez le logiciel

Dans le répertoire *Snake*, vous pouvez démarrer le jeu avec la commande :

```
$> ant Nibbles
```

Question 1: Evaluez la qualité du code logiciel

Installez *Sonar* et lancez le serveur comme expliqué dans l'introduction.

Dans le répertoire *Snake* :
– Compilez le projet avec la commande :
`$> ant build`
– Evaluez la qualité du code logiciel avec *Sonar* grâce à la commande :
`$> ant sonar`
– Accédez aux statistiques à l'adresse : `http://localhost:9000`

Vous utiliserez cet outil pour évaluer la qualité de votre code pour les questions suivantes.

La version "multi joueurs"

Nous voulons ajouter de nouvelles fonctionnalités pour mettre en oeuvre le mode multi-joueurs présenté en introduction.

Question 2: La gestion de la pomme

Dans la version initiale du logiciel, une nouvelle pomme est générée à chaque fois que le serpent en mange une. La gestion de la position des nouvelles pommes n'est toutefois pas satisfaisante, car la probabilité qu'une nouvelle pomme soit positionnée sur un carré utilisé par le serpent est non nulle.

Proposez une solution permettant d'assurer l'utilisation unique de chaque carré de l'arène. Ainsi, les nouvelles pommes ne pourront plus apparaitre sur un carré utilisé par le serpent.

Notez bien : vous n'avez absolument pas besoin de modifier les méthodes d'affichage. Ce travail consiste uniquement à ajouter de la synchronisation sur les carrés de l'arène et d'utiliser cette synchronisation dans les classes *Board*, *Apple* et *Snake*.

Pour faire ce travail :

1. Modifiez la classe *Board* afin qu'elle gère une liste synchronisée des carrés de l'arène :

- vous utiliserez une liste synchronisée contenant la liste des carrés utilisés (nous nous contentons de représenter un carré par son *Point* supérieur gauche) :
```
List<Point> lockedPoints = \
    Collections.synchronizedList(new LinkedList<Point>());
```
- vous implémenterez deux méthodes pour locker un carré :
```
synchronized Point lockPoint(int x, int y);
synchronized Point lockPoint(Point p);
```
- vous implémenterez une méthode pour unlocker un carré :
```
synchronized void releasePoint(Point p);
```

2. Modifiez les classes *Apple* et *Snake* afin qu'elles soient capables de s'approprier et de libérer les carrés de l'arène.
   Notez bien : les modifications des classes *Apple* et *Snake* sont minimes et consistent à s'assurer que le carré est libre avant de l'utiliser.

Compilez votre projet (**ant build**) et sa documentation (**ant doc**) ; auditer la qualité de votre code logiciel (**ant sonar**).

Question 3: Introduction du mode "multi joueurs"

Vous allez maintenant modifier le logiciel pour qu'il puisse y avoir plus de un serpent dans l'arène. Ce serpent devra se déplacer aléatoirement dans l'arène en respectant les règles de base énoncé dans l'introduction : s'il mange une pomme, il grandit ; s'il se mord ou se cogne aux bords de l'arène, le jeu prend fin.

La génération aléatoire du mouvement peut se faire de la manière suivante :

```
setCurrentDirection(Direction.fromInt(new Random().nextInt(1000)/250));
```

Pensez à faire le bon choix entre une classe qui emplémenterait `Runnable` ou une classe qui dériverait de `Thread`.

Compilez votre projet (**ant build**) et sa documentation (**ant doc**) ; auditer la qualité de votre code logiciel (**ant sonar**).