

Projet de Programmation 2

Aquarium distribué

Hugo Boujut, Simon Boyé, Cyril Cassagnes, Samuel Thibault

21 octobre 2011

L'objectif du projet est de réaliser en binômes un aquarium distribué. La composition des binômes devra être fournie le 28 octobre. De nouvelles versions de ce sujet (avec notamment de nouvelles idées d'extensions) seront éventuellement postées dans les news.

On rappelle qu'il s'agit d'un *projet de programmation orientée objet et réseau*, ce sont donc ces aspects qui seront évalués avant tout.

1 Sujet

1.1 Quelques liens utiles :

- Java : <http://java.sun.com/javase/6/docs/api/>
- Tutoriel interface swing : <http://zetcode.com/tutorials/javaswingtutorial/>
- Eclipse : <http://www.eclipse.org/>
- Export.jar : <http://www.fsl.cs.sunysb.edu/~dquigley/cse219/index.php?it=eclipse&tt=jar&pf=y>
- Modifier le classpath : <http://whitesboard.blogspot.com/2009/02/eclipse-classpath.html>
- Utiliser des commentaires javadoc : <http://www.siteduzero.com/tutoriel-3-35079-presentation-de-la-javadoc.html>

1.2 Description générale

L'objectif de ce projet est de réaliser un aquarium distribué : un ensemble d'objets qui évoluent (des poissons ou autres choses) dans un cadre fermé 2D (l'aquarium). Ces objets "vivent" sur des machines différentes, mais le rendu est commun. Cela signifie que lorsque l'on lance l'aquarium sur plusieurs machines, chaque machine fait évoluer ses propres objets, et affiche à l'écran non seulement ses objets, mais aussi les objets des autres machines, chaque machine devant donc envoyer le nouvel état de ses objets à toutes les autres machines pour qu'elles mettent à jour leur affichage. Si le programme est arrêté sur une des machines, les objets correspondant disparaissent de l'écran des autres machines. Si le programme est lancé sur une nouvelle machine, les nouveaux objets qui évoluent dedans apparaissent.

1.3 Information relative au code source

Le projet est packagé de la manière suivante :

```
'-- src
|  |-- aquarium
|  |  |-- factory
|  |  |  |-- AquariumItemFactory.java
|  |  |  |-- SeaweedFactory.java
|  |  |  '-- StoneFactory.java
|  |-- gui
|  |  |-- Animation.java
|  |  '-- Aquarium.java
|  |-- items
```

```

| | |-- AquariumItem.java
| | |-- MobileItem.java
| | |-- Seaweed.java
| | '-- Stone.java
| |-- move
| | '-- Mobile.java
| |-- Start.java
| '-- util
|     '-- RandomNumber.java
'-- image
    |-- algae.png
    '-- pierre.png

```

La classe Animation dérive de JFrame et la classe Aquarium dérive de JPanel. Cette dernière sera intégrée à Animation. La classe Start comporte le programme principale main() qui se contente d'instancier et de faire afficher une Animation. L'API Java comporte l'ensemble des informations concernant les opérations effectuées dans Start et Animation. La classe Aquarium contient les fonctions permettant de remplir l'aquarium ainsi que les fonctions graphiques. Ensuite nous avons définis la classe abstraite AquariumItem qui permet de créer des items dans l'aquarium. Aquarium contient les constantes NB_STONES et NB_SEAWEED définissant le nombre d'items. Par exemple, la classe Stone (pierre) et la classe Seaweed (algue) héritent de AquariumItem. Le constructeur de la classe Aquarium remplit l'aquarium en stockant les AquariumItem dans un attribut Collection<AquariumItem> items.

Pour l'instanciation des objets, nous utilisons des usines. La classe abstraite AquariumItemFactory<T extends AquariumItem> et la méthode public abstract T newItem() permettent d'avoir une usine d'objet générique. Comme par exemple la classe SeaweedFactory et la classe StoneFactory qui dérivent respectivement de AquariumItemFactory<Seaweed> et AquariumItemFactory<Stone>. La méthode newItem() retourne un T (Stone ou Seaweed selon) avec une position et une largeur aléatoires (largeur entre min et max). Nous Utilisons ces Factory dans fil de Aquarium en faisant "fabriquer" par exemple NB_STONES Stone.

Les Factory gèrent les collisions grâce à la méthode publique sink(Collection<AquariumItem> items, AquariumItem instance) dans AquariumItemFactory. Autrement dit chaque item doit trouver une place libre dans l'aquarium. cette méthode déplace l'objet instancié jusqu'à ce qu'il ait trouvé une place libre. Le constructeur d'Aquarium appelle newItem puis sink.

Pour peindre l'eau et tous les items nous utilisons gContext comme argument Graphics. La méthode paint se contente d'appeler drawImage(buffer, 0, 0, this); Mais pour que la méthode paint soit appelée, il faut terminer draw par un appel à repaint(); et pour que draw soit appelée, il faut terminer la méthode go() par un appel à draw(). Dans la méthode draw() chaque AquariumItem appelle drawImage sur son image. Pour dessiner, nous utilisons un buffer d'image, le dessin se fera dans le buffer avant d'être fait dans le JPanel ("double buffering").

Votre travail va consister à mettre un peu de mouvement dans votre aquarium. Il existe déjà une interface Mobile, comportant une méthode move, que la classe Aquarium.gui.Aquarium appelle pour donner aux objets l'occasion de se déplacer un peu à chaque *pas de temps*.

La classe abstraite MobileItem implémente alors cette interface : la méthode move appelle la méthode target (qui doit être fournie par la classe concrète qui en hérite) pour choisir une cible. Elle fait alors un petit déplacement en direction de la cible. La vitesse de déplacement est "inversement proportionnelle" à la largeur de l'objet.

```

Point target(Collection<AquariumItem> neighbours); // décider de sa destination en fonction
                                                    // de l'environnement

```

1.4 Prise en main sur une seule machine

Pour commencer de manière simple, on traite d'abord le cas d'une seule machine : pas de socket réseau, il s'agit de prendre en main le fonctionnement de l'aquarium.

1. Repérez dans le constructeur de la classe Aquarium et alentours l'ajout des pierres et algues. Créez une classe Fish qui hérite de MobileItem et la Factory correspondante; mettez des poissons dans l'aquarium.

2. Observez le fonctionnement de l'animation : dans la fonction `main` on a créé un thread `Time`, qui se contente d'appeler `aquarium.go()` en boucle puis attendre un peu (100ms). C'est `go()` qui s'occupe de l'affichage. Elle s'occupe également d'appeler la méthode `move` de chaque objet : si l'objet n'est pas arrivé à sa destination, on le fait bouger un peu dans sa direction.
3. Adaptez les classes pour que les poissons bougent : la classe `Fish` doit maintenant hériter de `MobileItem` plutôt que `AquariumItem`. Il faut donc y adjoindre une méthode `target`, qui pour l'instant choisit simplement un point au hasard. Observez que les poissons changent effectivement de destination dès qu'ils ont atteint celle qu'ils visaient.
4. Adaptez les classes pour créer un autre genre de poisson : toutes les secondes, le poisson change d'avis et choisit de nouveau une destination aléatoire. Il sera intéressant d'utiliser un thread pour cela.
5. Dessiner le diagramme des classes actuel. Il faudra le mettre à jour au fur et à mesure des évolutions du code.

2 Passage en réseau

Il s'agit maintenant d'afficher les poissons des programmes tournant sur d'autres machines.

Dans un premier temps, faites une version simple avec un support pour seulement 2 machines à la fois. Sur une machine on lance le programme sans argument, et le programme doit alors ouvrir une socket TCP en écoute. Sur l'autre machine on lance le programme avec le nom de la première machine en paramètre, le programme doit dans ce cas se connecter en TCP à la machine passée en paramètre.

Maintenant que l'on a une socket établie entre les deux programmes, il s'agit de transmettre les informations sur les objets. Il faut donc établir un protocole pour les exprimer. Il est recommandé d'établir un protocole textuel, qu'il sera ainsi facile de déboguer à la main. Réfléchissez et implémentez dans un premier temps votre propre protocole. Dans un deuxième temps, les groupes pourront mettre en commun leurs idées sur le TitanPad suivant :

<http://titanpad.com/ZtBNDYfDnY>

et converger vers un protocole standard que tout le monde utilisera, ce qui permettra de vérifier l'interopérabilité de vos projets. Attention bien sûr à établir un protocole qui permet d'ajouter des extensions tout en gardant la compatibilité avec les projets qui n'implémentent pas ces extensions.

Le principe du protocole est que chaque machine reste maître du mouvement des objets qu'elle gère. Les autres machines se contentent d'afficher ces objets aux nouvelles positions, lorsqu'elles les reçoivent via la socket. Il sera facile de vérifier que les affichages sont bien exactement les mêmes (au délai de latence près).

On utilisera des couleurs différentes pour distinguer les poissons venant de machines différentes.

3 Extensions

Une fois cette version de base développée, de nombreuses extensions sont possibles et intéressantes à développer. Il n'est pas obligatoire de toutes les développer pour avoir une bonne note, mais cela y contribue bien sûr fortement, il faut en développer au minimum quelques-unes. La liste n'est également pas exhaustive. Si vous avez des idées, discutez-en avec votre chargé de TD.

3.1 Serveur

Pour passer à plusieurs machines, le plus simple est d'utiliser un serveur centralisé, en réutilisant une version basique de votre serveur de discussion, qui va simplement diffuser les messages du protocole : une fois le serveur lancé, il suffit de dire aux différents programmes de se connecter à celui-ci.

3.2 À table !

Lorsque qu'un poisson piscivore rencontre un poisson plus petit que lui, il le mange et grandit un peu. Créez une nouvelle classe de poisson qui cible la position d'un poisson plus petit que lui. Ajoutez

des méthodes pour que lorsqu'il l'atteint, le poisson mangé est détruit. Étendez le protocole réseau pour que cela fonctionne aussi en réseau !

3.3 Gestion de collision

Si un poisson rencontre une pierre, il meurt.

3.4 Le cycle de la vie

3.4.1 Première version

Lorsque deux poissons de sexe opposé se rencontrent, ils font des petits, qui grossissent petit à petit jusqu'à une taille aléatoire. Cela doit bien sûr aussi fonctionner en réseau, les machines possédant père et mère créant chacune quelques petits.

3.4.2 Version ovipare

La plupart des poissons sont en fait ovipares : la mère dépose d'abord des $\frac{1}{2}$ ufs, qui doivent être fécondés plus ou moins rapidement par un père, avant de pouvoir grandir. D'un point de vue réseau, on dira que la machine possédant la mère distribuera quelques $\frac{1}{2}$ ufs à d'autres machines, aléatoirement.

3.5 Le cycle de la mort

Changez les dessins : les pierres sont remplacées par des étoiles, les algues par des planètes, et les poissons par des vaisseaux. Maintenant, les vaisseaux tirent des missiles qui, avec de la chances, atteignent leur cible.

3.6 Serveurs distribués

Pour éviter d'avoir un seul serveur centralisé, on peut étendre le serveur pour qu'il sache se connecter de lui-même à un autre serveur¹, dont le nom est simplement tapé sur l'entrée standard du serveur.

Ainsi, on peut avoir plusieurs "îlots", chacun comportant quelques programmes connectés à un même serveur, et les serveurs sont connectés les uns aux autres pour que les messages restent bien propagés de manière globale. Ainsi, si un des serveurs tombe, seuls les clients de ce serveur perdent leur connexion et peuvent se reconnecter à un autre serveur. Éventuellement le graphe d'interconnexion n'est alors plus connexe, car le serveur faisait "relai" entre deux autres serveur, mais ces deux autres serveurs peuvent se connecter l'un à l'autre pour rétablir la connectivité.

Pourquoi faut-il faire attention à ne pas introduire de boucle dans ce graphe de connexions ?

3.7 Gestion Acentrée

Intégrez le code du serveur dans le client. Lorsqu'un client se connecte à un autre client, il en récupère une liste de l'ensemble des machines actuellement inter-connectées. Si cet autre client quitte, le premier client peut ainsi essayer de se reconnecter à d'autres clients.

4 Organisation

Le projet est travaillé et étudié en binôme mais la notation est individuelle. Le projet *doit* être réalisé avec un outil de gestion de révision (svn, git, ...) : directement dans votre *home* ou sur la savanne du CREMI (<https://services.emi.u-bordeaux1.fr/projet/savane/>). Les enseignants évalueront la contribution de chacun des éléments du binôme au travail commun. Les enseignants se réserveront la possibilité de modifier la composition de chacun des binômes. Afin de permettre un travail profitable, il est conseillé de ne pas créer de groupes avec des niveaux trop différents.

1. à la manière du *peering* IRC

5 Rapport

Il s'agit de mettre en valeur la qualité de votre travail à l'aide d'un rapport. Pour cela le rapport doit explicitement faire le point sur les fonctionnalités du logiciel (lister les objectifs atteints, lister ce qui ne fonctionne pas et expliquer - autant que possible - pourquoi). À cet effet, on proposera des jeux de tests permettant de mettre en valeur la correction du logiciel (est-ce qu'il fait bien ce qu'on attend de lui?).

Ensuite le rapport doit mettre en valeur le travail réalisé sans paraphraser le code, bien au contraire : il s'agit de rendre explicite ce que ne montre pas le code, de démontrer que le code produit a fait l'objet d'un travail réfléchi et même parfois minutieux. Par exemple, on pourra évoquer comment vous avez su résoudre un bug, comment vous avez su éviter/éliminer des redondances dans votre code, comment vous avez su contourner une difficulté technique ou encore expliquer pourquoi vous avez choisi un algorithme plutôt qu'un autre, pourquoi certaines pistes examinées voire réalisées ont été abandonnées.

Il s'agira aussi de bien préciser l'origine de tout texte² ou toute portion de code empruntée (sur internet, par exemple) ou réalisée en collaboration avec tout autre binôme. Il est évident que tout manque de sincérité sera lourdement sanctionné.

Pour conclure on pourra traiter des limites et extensions possibles des logiciels proposés. Enfin on présentera une bibliographie (livres, articles, sites web) brièvement commentée. Ce rapport est le témoin de vos qualités scientifiques mais aussi de vos qualités littéraires (style, grammaire, orthographe, présentation). Pour présenter votre logiciel, on pourra adopter le plan suivant :

1. Présentation des fonctionnalités
2. Valorisation du travail réalisé
3. Diagramme des classes
4. Conclusion
5. Bibliographie commentée
6. Annexes et code du projet

6 Soutenance

La présentation finale du projet se fera en salle de TD autour d'une démonstration, des questions individuelles pourront être posées.

2. Directement dans le texte, par une note en bas de page comme celle-ci ou par une référence bibliographique entre crochet [1].