TP 7 : interfaces, classes génériques, collections

Partie I

Dans les TP précédents, à chaque fois que vous avez implanté une simulation, vous avez dû réimplanter une petite application pour visualiser son évolution. Nous allons commencer par rendre le code plus modulaire en extrayant une application réutilisable. Pour l'instant, cette application mélange encore trois éléments : la vue, le contrôleur, et l'application elle même. Nous verrons en janvier comment séparer ces éléments pour plus de modularité.

Exercice 1 (Une application textuelle pour terrain 1D)**.**

- Consultez les classes `Terrain1D`, `Terrain1DApp` fournies.
- La classe `Terrain1D` est celle du TP3. Comment a été modifiée la classe `Terrain1DApp` par rapport au TP3 ?
- Quelle est le rôle des deux classes ?
- Essayez-les.
- Remplacez la classe `Terrain1D` par la vôtre. Vérifiez que tout fonctionne.

Exercice 2 (Une application textuelle réutilisable)**.**
On souhaite réutiliser la même application textuelle pour visualiser l'évolution d'un des terrains implanté lors du TP 6 (évolution d'une population de mulot, de souris, ...).

- Ajoutez une méthode `toString` à `Terrain` (elle pourra par exemple afficher le nombre d'êtres vivants).

  Si votre classe `Terrain` n'est pas fonctionnelle, vous pouvez utiliser à la place la classe `SystemeDynamiqueExemple`.
- Premier essai : dupliquez le code de `Terrain1DApp` et adaptez-le ; qu'avez-vous dû changer ?
- Qu'est-ce qui est dupliqué ?
- Un code dupliqué, *cela sent mauvais*. Maintenant que vous avez appris ce qu'il avait à vous enseigner, jetez le.
- Consultez l'interface `SystemeDynamique`. Un système dynamique est un objet avec une méthode `evolue` et, comme tout objet en Java, une méthode `toString`.
- Adaptez les classes `Terrain1D` et `Terrain` pour qu'elles implantent l'interface `SystemeDynamique` (`class Terrain1D implements SystemeDynamique`).
- Extrayez une classe `SystemeDynamiqueApp` de la classe `Terrain1DApp`.
- Adaptez la classe `Terrain1DApp` pour utiliser `SystemeDynamiqueApp`. Créez de même la classe `TerrainApp`.

Exercice 3 (Une application « graphique » réutilisable)**.**

- En vous inspirant de l'exercice précédent, extrayez une classe `SystemeDynamiqueGUI` de `Terrain1DGUI` pour obtenir une classe `TerrainGUI` sans duplication.

– (optionnel) dans le main de `Terrain1DGUI`, construisez plusieurs terrains ainsi qu'autant d'applications pour visualiser l'évolution de ces terrains. Vous lancerez ensuite ces applications (avec la méthode `run`) dans des `Thread`s séparés.
– (optionnel) Que se passe-t'il si on lance plusieurs applications sur le même terrain ?

Exercice 4 ((optionnel) Classes génériques)**.**
On souhaite maintenant arriver au même but que précédemment (avoir une application textuelle/graphique réutilisable), mais sans avoir à modifier les classes `Terrain1D` et `Terrain`. En particulier on ne pourra pas leur faire implanter une interface commune (situation typique, hélas : vous n'avez que le bytecode de ces classes et pas leur sources).
– Transformez la classe `Terrain1DApp` en une classe générique `App<SystemeDynamique>`.
– Utilisez-la pour implanter `Terrain1DApp` et `TerrainApp` sans duplication.
– Avantages et inconvénients de cette approche ?

Partie II

Jusqu'à la fin de la semaine, il n'y aura plus que du TP. Le but est de mettre en pratique tout ce que vous avez appris jusqu'ici, à l'échelle d'une plus grosse application.

L'objectif est d'implanter une simulation SimSavane 3.0, avec un terrain à une puis deux dimensions (comme dans le TP2) dont les êtres vivants sont modélisés par des objets (comme dans le TP 6). Les êtres vivants pourront, par exemple, se reproduire, mourir, manger, chasser, vieillir, etc. À vous, en tant que « spécialiste métier », de définir les règles d'évolution précises. On pourra faire des hypothèses simplificatrices comme « Une case contient un unique être vivant. Un autre être vivant ne peut aller sur une case déjà occupée qu'en mangeant son occupant ».

À vous, en tant qu'« informaticien(ne) », de définir les étapes intermédiaires. Vous préciserez à chaque fois le cahier des charges, un diagramme UML des objets en jeu et un protocole de test approprié pour vérifier que ces objectifs intermédiaires sont bien remplis.

Lorsque vous rencontrerez des bogues, une fois corrigé prenez du recul pour voir si vous auriez pu l'éviter (par exemple avec des spécifications ou des tests plus précis).

Vous ferez à la fin de chaque étape une copie de votre projet. Pour cela, si vous savez faire, le mieux est d'utiliser un système de gestion de versions comme SVN, GIT, ou Mercurial. Sinon faites une copie à la main dans un répertoire « etape1 » (puis « etape2 », ...).