ISI 3

tp4. rEFACTORING de code & MVC

*Enseignant : Matignon Laëtitia*

POLYTECH LYON

Domaine Scientifique de La Doua – 15, Boulevard Latarjet

69622 VILLEURBANNE CEDEX

Tél. (33) 04.72.43.12.24 - Fax. (33) 04.72.43.12.25

Web: http://polytech.univ-lyon1.fr/

# Refactoring du code et MVC

Notre application traite de la gestion de plusieurs tortues et de leurs déplacements sur une feuille de dessin. Pour cela, nous utilisons le patron d’architecture MVC afin de séparer tous les traitements (Contrôleur), les données (Métier) et la présentation (Vue).

Afin de faire communiquer notre modèle avec d’autres objets, nous allons utiliser le pattern observer. Notre fenêtre principale (la vue) implémente l'interface **Observer**, et référence le modèle qui hérite de la classe **Observable**. La vue observe donc le modèle, qui à chaque mise à jour remonte l'information aux observateurs par le biais de la méthode **notify()**. La classe Controler elle, s'occupe de réceptionner les entrées utilisateur et applique les traitements au modèle.

De manière concrète, dès que nous voulons ajouter une nouvelle Tortue normale non autonome), nous avons ces classes internes qui réceptionnent le clic de la souris à l’aide de ***listeners***, ordres qui sont transférés au contrôleur qui va ensuite demander au modèle de ce mettre à jour et ainsi modifier la vue.

*//Implémentation du pattern observer*

*public class Modele extends Observable {*

*public List<Tortue> tortues;*

*…*

*}*

A chaque changement du modèle on appelle la méthode notifyChanges() :

*// Dans la classe Modele*

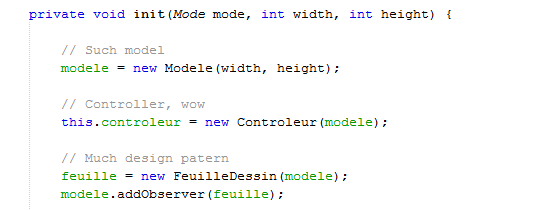
*public void notifyChanges() {*

*setChanged();*

*notifyObservers();*

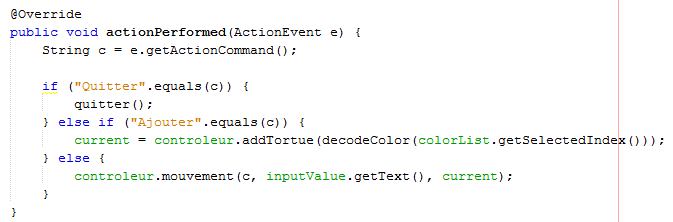
*}*

Dans notre vue, nous avons donc une méthode d’initialisation avec le modèle à observer.



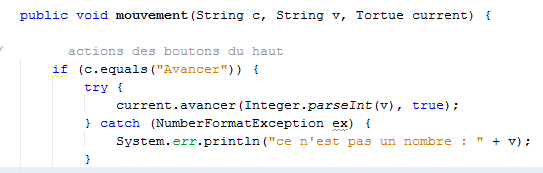
*Figure 1 : Méthode init de notre vue*

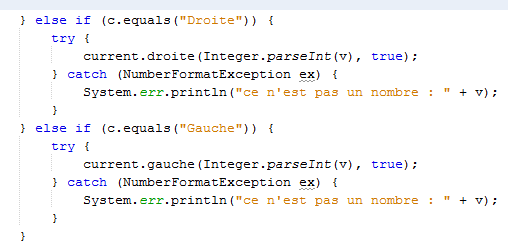
Notre conception doit aussi permettre d’ajouter facilement d’autres formes pour les tortues (cercles, polygones..). De ce fait nous avons nos différentes formes de tortues qui hériteront de notre classe abstraite GraphicTurtles.

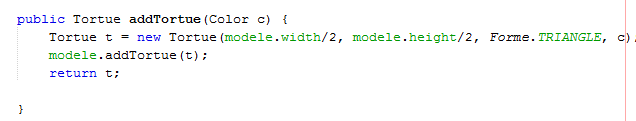
Les différents listeners : 

*Figure 2 : Les listeners (actionPerformed) de notre vue*

Notre contrôleur transforme ces ordres (mouvements, ajouts de tortues normales) et va dire au modèle quelles méthodes utiliser :

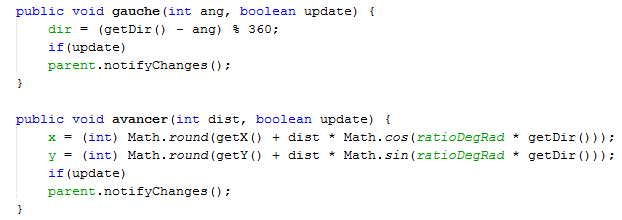




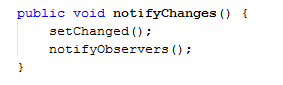


*Figure 3 : Récupération des commandes utilisateurs par le contrôleur*

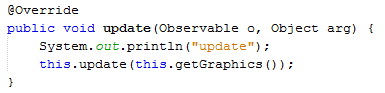
Puis le modèle effectue les différentes actions et va demander à la vue de se mettre à jour.



*Figure 4 : Le modèle effectue les différentes actions*

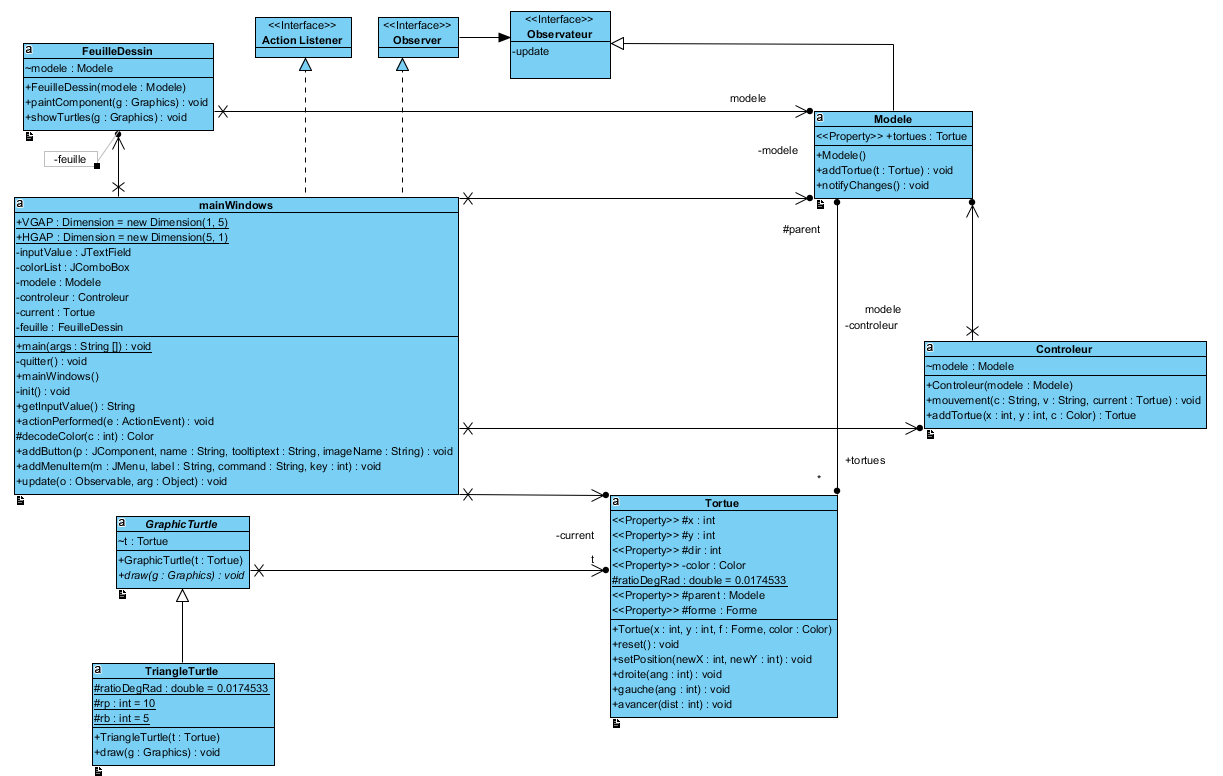


*Figure 5 : Le modèle rafraîchit la vue*



*Figure 6 : Le vue se rafraîchit via la méthode update.*

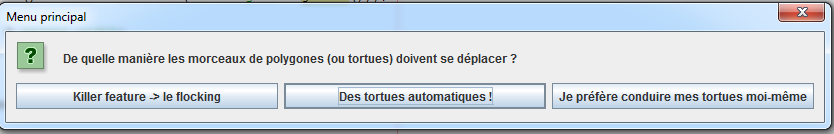
Nous obtenons donc, avant le refactoring, le diagramme UML suivant :



*Figure 7 : Diagramme UML après refactoring*

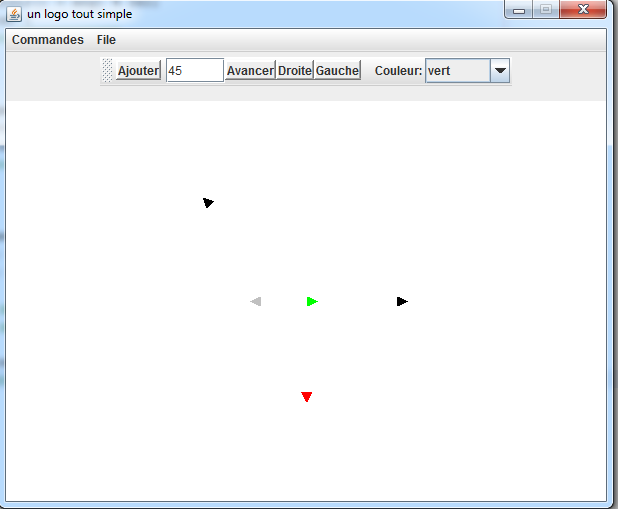
# Extension de l’application.

Si on compile le projet, nous obtenons le choix entre nos trois types de mode : manuel, automatique ou flocking.



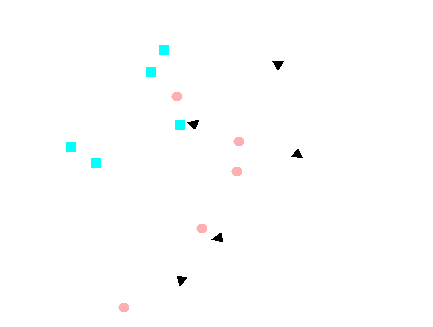
*Figure 8 : Choix du mode de lancement des tortues*

Tortues manuelles :



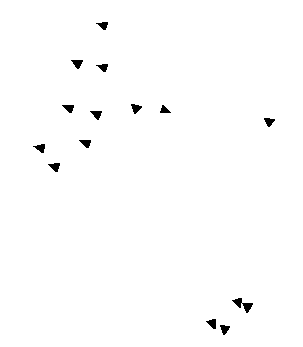
*Figure 9 : Tortues en mode manuel*

Tortues Automatiques :



*Figure 10 : Tortues en mode automatique*

Tortues en mode flocking :



*Figure 11 : Tortues en mode flocking*

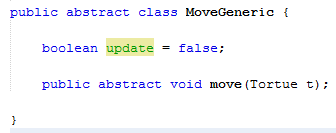
# Patrons de conceptions utilisés

### Observable /Observer

Comme expliqué précédemment, nous avons utilisé le patron de conception Observable/Observer. La vue contient une liste d’observateurs pour chaque item, chaque bouton. Le clic sur un item de l’IHM est transmis au contrôleur qui va ensuite dire au modèle quelles méthodes utilisées (avancer, droite, gauche…) et ensuite rafraîchir la vue (grâce au *setChanged()* notamment).

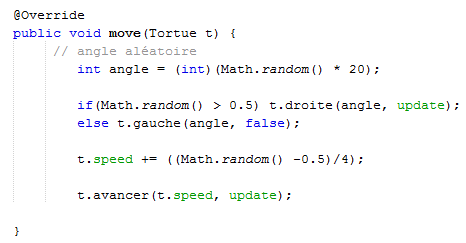
### Stratégie

Nos différents types de Tortues se déplacent de manière différente : manuellement, de manière automatiques ou par flocking. Mais si demain nous voulons rajouter un nouveau moyen de déplacement, il convient d’utiliser le design pattern **Stratégie**. Avec ce design pattern, nous avons notre classe générique avec la méthode de déplacement :



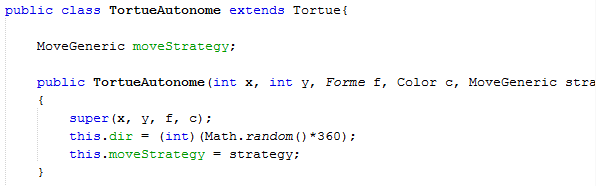
*Figure 12 : Patern Stratégie*

Ainsi, toutes nos classes de déplacement héritent d’un algorithme de déplacement personnalisé. Pour une nouvelle tortue, nous pouvons donc lui définir un certain type de déplacement, voici par exemple l’algorithme de déplacement pour la classe MoveAutonome().



*Figure 13 : méthode move () des tortues autonomes*

A la création de la tortue, on lui instancie une classe de déplacement. Lorsque la méthode de déplacement est appelée, c'est le code de cette classe de déplacement qui est exécuté.

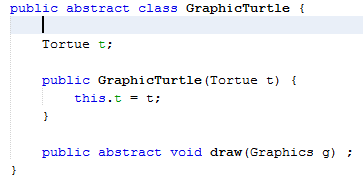


*Figure 14 : Instanciation de la stratégie*

### Factory et Singleton

Dans notre application, nous pouvons instancier plusieurs formes de tortues : des tortues rondes, triangles ou carrées.

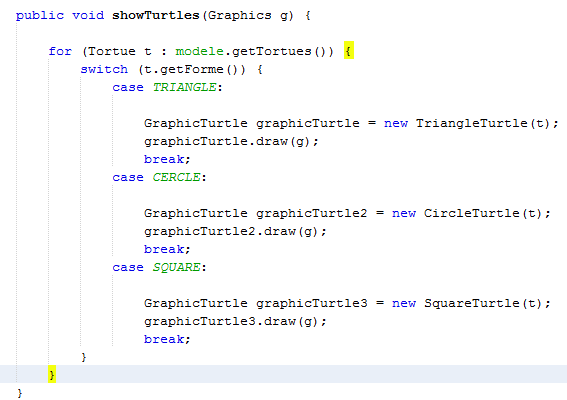
Chaque classe de dessin de Tortue hérite d’une classe GraphicTurtle :



*Figure 15 : Class abstraite GraphicTurtle*

Chaque tortue à sa propre méthode draw selon ses spécificités.

Actuellement, l’affichage de ces tortues est géré par la vue par un switch :

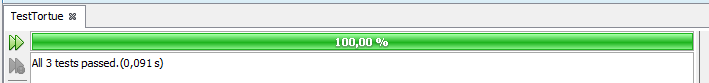


*Figure 16 : Affichage gérée par la vue*

Il était en effet possible de gérer cet affichage dans une classe séparée, dans une optique de « separation of concerns ». Cette gestion permet l’implémentation du design pattern Factory (et Singleton pour indiquer qu’on aura une Factory unique). Il ne s’agit donc que de déplacer les actions gérées actuellement par la vue dans une classe autre. Le design pattern n’est donc pas implémenté actuellement mais cette fonctionnalité peut être ajoutée rapidement.

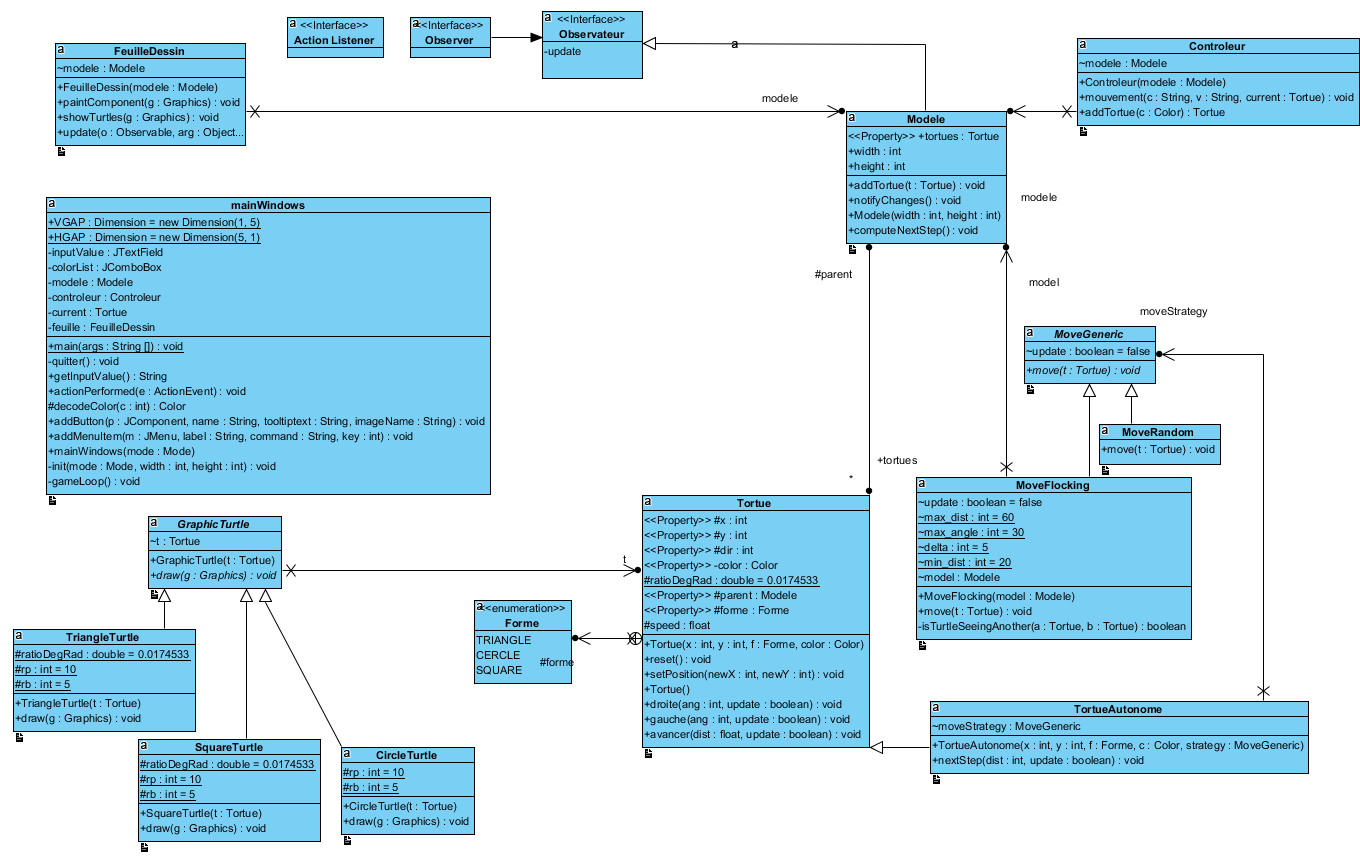
# Tests unitaires.

Des tests unitaires ont été effectués sur les différentes méthodes de notre classe Tortue.java.



*Figure 17 : Tests unitaires*

Voici le diagramme de classe final :



*Figure 18 : Diagramme de classe UML final*