

Compte rendu TP4

ISI 3 – Refactoring de code & MVC

LETOurneur leo - Gerland Loïc

POLYTECH

9

Juin

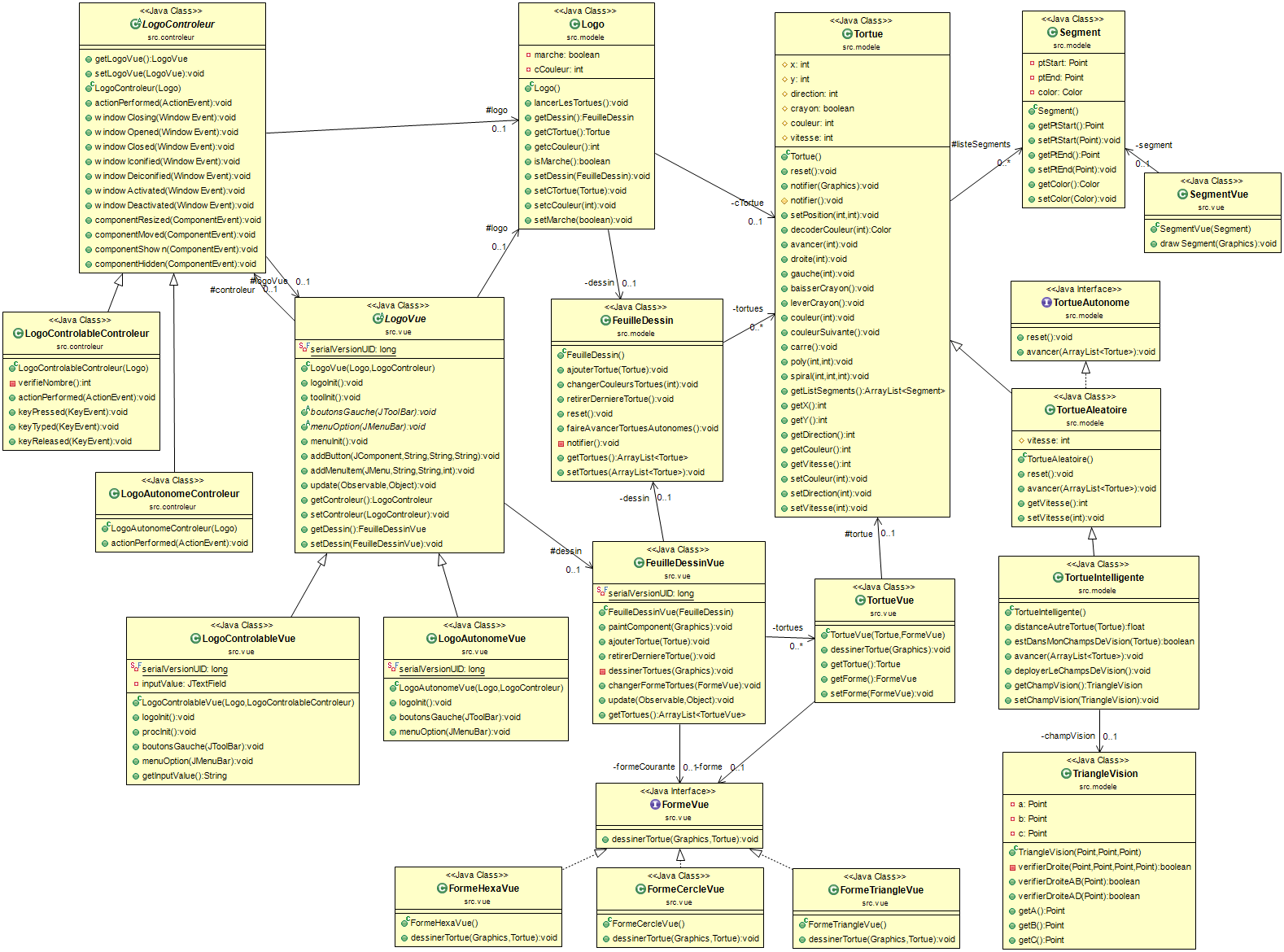
2016

# Introduction

Dans ce TP, nous partions d’un projet initial logoInit qui était composé de trois classes :

* Tortue : Classe représentant une tortue qui peut se déplacer sur une feuille en dessinant un trait sur son passage. Cette classe contient également une classe Segment qui représente le trait
* Feuille Dessin : Classe représentant la feuille sur laquelle la tortue dessine. Elle possède une liste de tortue en attribut et hérite de la classe JPanel pour l’affichage.
* SimpleLogo : C’est la classe principale du programme, elle représente la fenêtre principale. C’est elle qui lance le programme et instancie la feuille dessin ainsi que la tortue. On retrouve dans ce classe tous les ActionListener qui vont effectuer diverses actions en fonction des clics de l’utilisateur.

Ce projet fonctionnait en l’état mais il n’était pas modulaire et ne respectait pas l’architecture MVC. Le but de notre projet était donc d’effectuer le refactoring du code afin de mettre en place une architecture MVC et répondre à différentes questions. Au final, nous avons mis en place une architecture que vous pouvez voir dans le diagramme de classe ci-dessous :



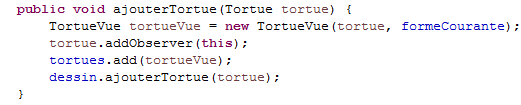
Nous allons maintenant expliquer plus en détails ce diagramme en justifiant les choix des patrons de conception en fonction des besoins.

## Architecture MVC

Pour mettre en place une architecture **MVC** sur ce projet, nous avons tout d’abord créé trois packages (modèle, contrôleur et vue) afin de séparer les différentes classes en fonction des couches. Ensuite, nous avons repris le code existant et l’avons replacé dans les packages correspondant à leur fonction. Pour cela nous avons découpé chacune des classes de départ en deux classes distinctes, une classe modèle qui contient tous les traitement métier de la classe et une classe vue indépendant de la première qui contient tous le code correspondant à l’affichage de l’objet.

Par exemple pour la tortue, nous avons créé une classe TortueVue qui contient la méthode dessinerTortue et une classe Tortue qui représente le modèle de la tortue avec toutes les fonctions qui vont modifier l’état de la Tortue, à savoir sa position ou sa couleur.

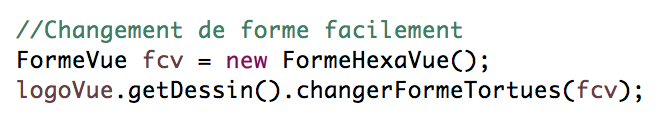
Ensuite pour faire la liaison entre la vue et le modèle, nous avons utilisé le patron de conception **Observateur/Observé**. Dans notre cas, c’est le modèle qui est un observable et la vue qui observe le modèle. Cette liaison est faite au moment de la création de la vue qui prend en paramètre le modèle.

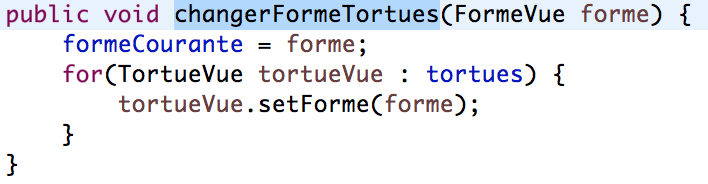


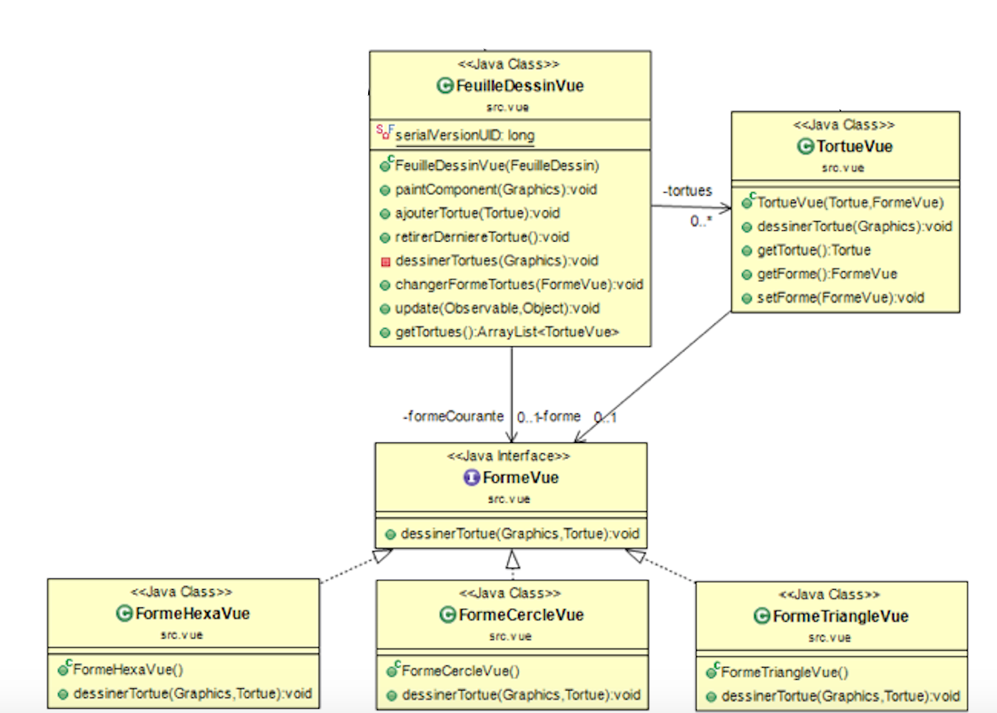
Vous pouvez voir dans le code ci-dessus un exemple de cette liaison dans le cas des tortues. Cette fonction est appelée dans le FeuilleDessinVue par le contrôleur lorsque l’utilisateur cliquer sur le bouton « Ajouter ».

## Nouvelles formes de tortue

En modifiant le code, nous devions créer un moyen simple de changer la forme de la tortue depuis l’extérieur du programme (principe ouverture/fermeture SOLID). Nous avons pour cela utilisé un pattern **Stratégie** en insérant un attribut forme de type FormeVue (interface) dans la TortueVue. Il est donc maintenant facile d’ajouter des formes indépendantes en implémentant simplement l’interface FormeVue et en modifiant l’attribut dans FeuilleDessinVue qui va modifier toutes les tortues. La méthode dessinerTortue() sera appelée directement dans la forme courante.







Pattern Stratégie

## Nouvelle fenêtre

Une fois le premier refactoring effectué, il nous a été demandé de créer un nouveau modèle de tortue qui se déplace aléatoirement dans un environnement toroïdal. Pour cela avant de penser à la tortue, nous avons créé une nouvelle fenêtre car ici l’utilisateur ne peut pas intervenir sur les déplacements de la tortue. Nous avons décidé de rendre LogoControleur et LogoVue abstraites, pour pouvoir faire la différence entre une fenêtre contrôlable et une fenêtre aléatoire. En effet, les implémentations de listener et les éléments de la vue ne sont pas les mêmes. Le modèle était cependant identique.

|  |  |
| --- | --- |
| ../../../Desktop/Capture%20d’écran%202016-06-10%20à%2000.09.30.png | ../../../Desktop/Capture%20d’écran%202016-06-10%20à%2000.09.41.png |

Nous avons pu mettre en place un pattern **Patron de méthode** dans l’héritage de la vue et plus précisément avec la méthode logoInit() qui utilise les méthodes toolInit() et menuInit(), qui utilisent à leur tour, les méthodes boutonsGauche() et menuOption() définies dans la vue abstraite en tant que fonctions abstraites et redéfinies dans les classes concrètes :

|  |  |
| --- | --- |
| ../../../Desktop/Capture%20d’écran%202016-06-07%20à%2010.01.19.png | ../../../Desktop/Capture%20d’écran%202016-06-07%20à%2010.01.38.png |
| ../../../Desktop/Capture%20d’écran%202016-06-07%20à%2010.01.29.png | ../../../Desktop/Capture%20d’écran%202016-06-07%20à%2010.01.44.png |

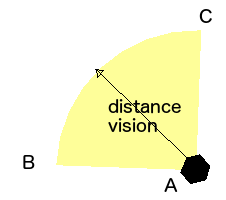
## Tortues Aléatoires

Afin de proposer une architecture modulaire nous avons décidé d’utiliser un héritage et une interface pour la création de nouveau type de Tortue. Le nouveau type de tortue, doit hériter de la classe Tortue afin de récupérer les attributs et fonctions nécessaire à la définition d’une tortue. Ensuite si l’on souhaite créer un type de tortue autonome, il faut implémenter l’interface TortueAutonome et ainsi déclarer la fonction avancer(ArrayList<Tortue>) qui prend en paramètre la liste des autres tortues.

Ainsi pour notre classe TortueAleatoire, il nous a suffi définir la fonction de déplacement de l’interface en prenant en compte les limites de la feuille afin de replacer la Tortue et obtenir un environnement toroïdal.

## Tortues Intelligentes

Les tortues intelligentes utilisent la même fonction de déplacement que les tortues aléatoires, à la différence qu’elles déploient un champ de vision pour voir si des tortues ne sont pas à portée de vue. Si une tortue aperçoit d’autres tortues, elle prend la direction et vitesse moyenne de ces tortues (en s’incluant dans l’équation). Le champ de vision est calculé de la manière suivante : Calcul du point B et du point C, pour pouvoir tracer les droites AB et AC. Pour chaque autre tortue, on vérifie que la tortue est entre ces points (si les coordonnées de la tortue sont du même côté que C en séparant le plan par la droite AB, et que les coordonnées de la tortue sont du même côté que B en séparant le plan par la droite AC, et enfin que la distance séparant les deux tortues est plus petite que la distance de vision.



# Conclusion

Pour conclure, ce TP nous a permis de réviser l’architecture MVC en la mettant en place sur un projet existant. Nous avons également pu appliquer différents patrons de conception vu en cours afin de répondre aux différents problèmes du sujet et rendre le projet le plus modulaire possible. Au cours du projet, nous avons dû faire face à plusieurs problèmes qui nous ont demandé de repenser notre architecture. Notamment lors de la dernière question car il fallait que la FeuilleDessin puisse connaître les informations sur chaque tortue afin de les faire converger vers une même direction lors d’éventuelles croissements.