UNIVERSITÉ
CÔTE D'AZUR

Université Côte d'Azur

Takenoko
https ://github.com/Mahe-Thomas/takenoko

Auteur :
Jeremy Bonsaudo
Matthias Percelay
Brandon Fontany–Legall
Thomas Mahe

6 novembre 2018

Table des matières

Chapitre 1

Présentation du projet

1.1 Sujet

Le projet consiste à réaliser en Java une version numérique du jeu Takenoko créé par Antoine Bauza. Version textuel n'étant pas destinée à être jouée par des êtres humains mais par des robots jouants de façon autonome. Dans le Takenoko nous allons endosser le rôle d'un jardinier japonais affairé à répondre au mieux aux requêtes du vénérable empereur. Pour ce faire, nous aurons, durant notre tour de jeu, deux choses à faire : Regarder quel temps il fait et effectuer deux actions parmi les possibilités suivantes : agrandir le jardin, irriguer, bouger le jardinier, bouger le panda.

1.2 Avancement du projet

L'intégralité des fonctionnalités du jeu sont implémentés à l'exception du dé météo. Bien que les joueurs le lancent au début de leur tour, cela n'influe pas sur le déroulé du tour.
Cependant une règle n'est pas respectée par les robots, ils peuvent effectuer plus de deux actions par tours. Nous estimons que deux itérations supplémentaires auraient été nécessaire pour terminer le moteur selon les règles du jeu.

1.3 Problématique soulevée

1.3.1 Détecter et réduire la dette technique

L'organisation du code a été pour nous un assez grand problème notamment à cause de l'apparition de "God class" [1]
comme la classe de Jeu qui concentré un trop grand nombre de fonctions. Nous avons donc fait plusieurs réaménagements *Voir paragraphe Conception*
A l'inverse des "God Class", la méthode agile nous impose de concevoir le projet comme un ensemble de petites briques. Durant la vie du projet il peut s'avérer que plusieurs briques ont des besoins fonctionnels similaires, l'ajout de code générique devient alors nécessaire pour répondre aux bonnes pratique de conception orientés objets. Ces phases d'ajout de généricité n'ajoute aucune plus value au projet d'un point de vue client mais doit être vu comme un investissement pour le développement futur.

1.3.2 L'injection de dépendance

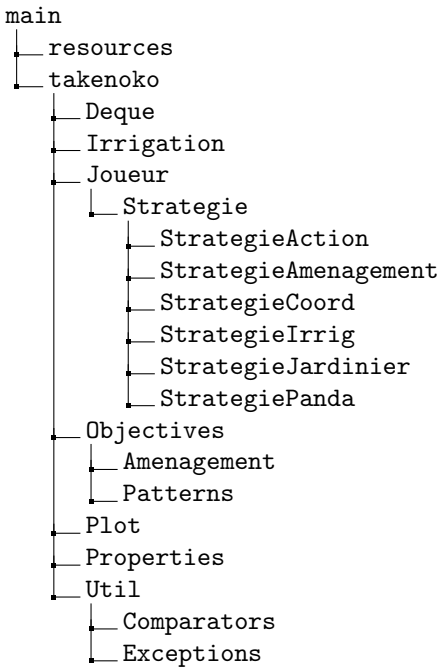
Du fait de la nouveauté de l'outil et de l'avancée du projet, nous avons rencontrer plusieurs problèmes lors de l'injection de dépendance.
Bien que la détection des classes nécessitant des injections de dépendance soit "facile", l'ajout d'IoC sur une classe à un impact fort sur les classes l'utilisant. Cela implique de réviser l'ensemble des tests et d'adapter le reste du projet. *Voir paragraphe Spring*

1. La "God Class" est une classe comportant un très grand nombre de fonctions qui pourraient être dispatcher dans d'autre classes

Chapitre 2

Organisation du code

2.1 Packages



Après de multiples ré-aménagements, nous sommes arrivé à cette structuration des packages.

2.2 Interfaces

Les principales Interfaces sont celle des Stratégies, chaque stratégie que ce soit StratégieAction, StratégieAmenagement, etc... ont chacune leur Interfaces car plusieurs implémentations avec des idées différentes sont présentes. Toutes ces interfaces permettant l'implémentation d'une stratégie complexe composée d'une stratégie de chaque catégorie.

2.3 Héritage

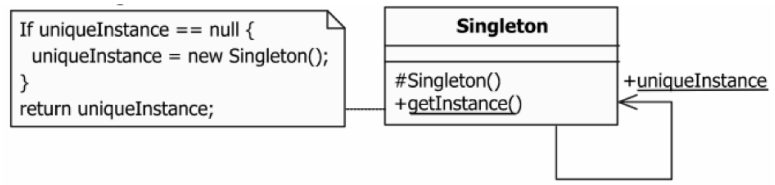
Nous utilisons l'héritage pour les éléments ayant comme dit précédement des besoins fonctionnels similaires, nous avons donc du code générique pour les pioches et les cartes objectifs. Les comportements du type pioche/validation d'une carte étant commun à l'essemble des cartes, il est important que les classes spécifique hérite d'une classe contenant les comportements primaires de ces éléments.

Chapitre 3

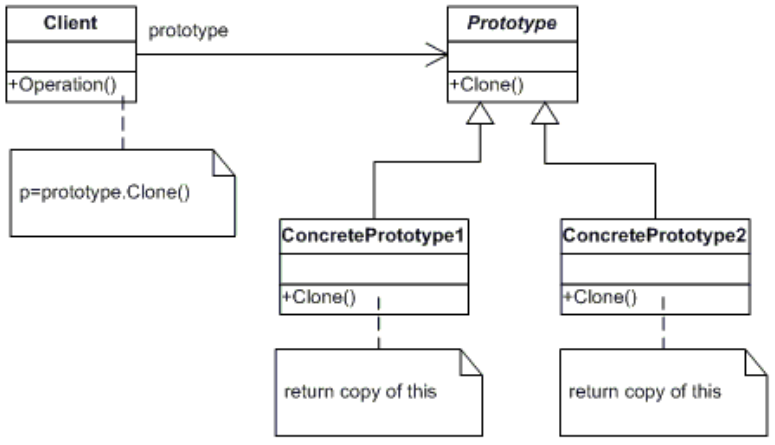
Conception

3.1 Patron de conception

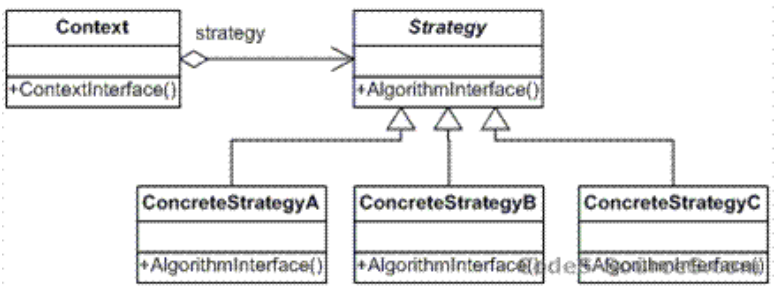
Singleton : Utilisé pour notre Logger, un unique objet est instancié lors du premier appel et ce pendant toute la durée de vie du programme.



Prototype : Utilisé pour la quasi totalité des beans Spring. Permet d'instancier plusieurs instances d'objets selon une unique spécification. Très utile pour générer les decks.



Strategy : Utilisé pour les stratégies de nos robots. Permet d'encapsuler plusieurs stratégies afin d'avoir une stratégie mère se chargeant de produire le résultat espéré selon une action défini.



Chapitre 4

Spring

Afin de prendre en main le framework Spring et la notion d'injection de dépendance, nous avons au préalable migré les classes de pioches. Instanciés alors via des définition Json, nous avons donc traduit ces définitions en Beans. Nous avons ensuite étendu l'ensemble des composants Spring du projet.
Il s'est avéré que la spécification d'object sous forme de bean (xml) est plus simple que les définition en Json. La récupération des objets est également simplifié par rapport à un parsing Json.

4.1 Notre Spring

4.1.1 Composants

— PlotsDeck : la pioche est créé à partir de définition de beans de Parcelles.

— ObjectivesDeck : Différentes pioches peuvent être créer selon le type de cartes objectifs (beans).

— Plateau : le plateau de base de takenoko est obtenu via un bean.

— Joueur : différents type de joueur peut être instancié via des beans avec pour chacun une stratégie qui lui est propre.

— Game : Une partie est paramétrable selon les composants cité ci-dessus.

— GameStarter : Peut être paramétré via les paramètres de l'application spring (nombre de parties / mode du Logger).

4.1.2 Contextes

Notre projet est pourvu de deux contextes, un pour l'exécution du programme et l'autre pour l'environnement de test. Cela nous permet de tester notre projet dans un environnement que l'on peut définir et qui peut donc être très différent d'une partie "normal" du jeu. Cela nous permet de tester le projet dans des situations plus inattendus.

4.1.3 Paramétrisation

L'ajout de l'injection de dépendance permet une grande flexibilité dans la configuration de nos parties et des composants qui la compose. Ainsi, sans toucher au code Java, on peut ajouter un deuxième type de partie où le nombre de joueurs ainsi que les stratégies de ceux-ci sont différentes.
Il est également plus facile de créer un grand ensemble de parties destinées à être joué par la suite en demandant simplement la création d'un objet selon le nom d'un bean.

Chapitre 5

Bilan

Projet dans l'ensemble plutôt abouti même si il reste encore pas mal de travail notamment avec l'ajout du dès météo qui est planifié pour une version prochaine. Après pas mal de restructuration du projet les package sont plus ou moins cohérent de notre point de vu mais en moins les stratégies restent difficiles à prendre en main pour un nouveau développeur arrivant sur le projet. De plus, les patrons de conception nous ont été d'une grande utilité notamment pour les stratégies et la conception d'une stratégie complexe dérivant d'une multitude de petites stratégies spécialisées. Pour finir, nous nous sommes heurté à la difficulté de Spring un peux tardivement ce qui nous obligea à modifier énormément de classes dans notre projet.

Pour conclure, le passage de Spring est un passage obliger au vu de la composition des classes notamment la Classe Game qui connect "les fils" entre les différentes fonctionnalités de l'application. Cependant, le passage à spring devrait dans l'idéale se faire dès le début du projet.