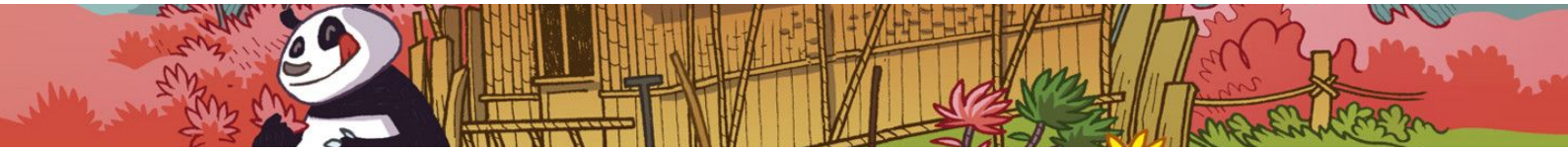
Takenoko

Loïc GERMERIE, Théo QUI, Cyril MARILIER, Violette HERON



Takenoko est un jeu de plateau créé par Antoine Bauza et sorti en 2011. Dans ce jeu, nous endossons le rôle d'un jardinier devant faire pousser des bambous dans le jardin impérial japonais, alors qu'un panda les mange en parallèle. Les joueurs sont assis autour d'un plateau qu'il doivent former eux-mêmes en y posant des tuiles de différentes couleurs, et peuvent y faire évoluer un jardinier et un panda. Le joueur a une liste d'objectifs à compléter : des motifs à créer en posant les tuiles colorées sur le plateau, des hauteurs de bambous à faire pousser, mais également des hauteurs de bambous à manger en prenant le rôle du panda.

Le projet qui nous a été donné ce semestre était l'implémentation de ce jeu, ainsi que de robots capables d'y jouer. Nous n'avions pas à implémenter d'interface graphique, seulement une sortie console suffisamment lisible pour pouvoir suivre le jeu.

L'organisation générale du code

Découpage en package et relations d'héritage

Dès le début du projet se sont clairement divisés trois parties dans notre code qui allaient devoir être distinctes, donc dans des packages séparés : le **moteur de jeu**, les **joueurs** et les **objectifs**, finalement contenus dans le package de moteur de jeu mais ayant leur fonctionnement propre au sein de ce package. Ce sont également ces trois packages qui nous ont servi de "couches" lors du découpage de nos itérations en tâches.

Nous avons ensuite remarqué que certaines parties de notre code pouvaient être implémentées différemment pour différentes utilisation, nous avons donc créé des interfaces pour ces classes-là. Par exemple, nous avons fait le choix d'implémenter notre plateau grâce à une HashMap, mais si ce choix s'était révélé peu pratique pour certains cas, il aurait été facile de changer cette implémentation de façon complètement transparente pour le reste du code. De même avec les tuiles, ce qui s'est révélé inutile et nous avons fait disparaître l'interface. Nous en dirons plus sur la hiérarchie des autres packages dans la partie concernant les design patterns intégrés.

Interactions entre les classes

Le plateau de jeu rend des services liés à sa disposition : où il est possible de poser une tuile, où il est possible de bouger les pions, où il est possible de poser une irrigation… Le joueur a donc connaissance de la board et de ses services pour pouvoir construire sa stratégie. Il a également besoin d'avoir des informations sur les objectifs qu'il tient en main,

et comme ce sont les objectifs eux-mêmes qui gardent l'information sur s'ils sont complétés, il pourra en prendre connaissance lors qu'il pioche un objectif.

Si nous avons essayé de garder le couplage entre les classes le plus faible possible, il y a cependant des cas où il ne nous est pas apparu d'autre solution. Par exemple, pour résoudre les objectifs de panda, nos objectifs doivent accéder à l'estomac d'un joueur et pouvoir le modifier. Au lieu d'ouvrir la modification de l'estomac à tout le jeu, nous avons fait en sorte que le jeu fasse un traitement particulier à la validation de ce type d'objectifs, ce qui renforce le couplage entre les objectifs et le jeu.

L'utilisation des patrons de conception

Pattern Template : les joueurs

Nous devions répondre aux problématiques suivantes : donner un cadre lisible à un développeur souhaitant développer un robot, mais verrouiller un flow de jeu commun à tous les joueurs. Ainsi, un pattern template s'est imposé pour les robots, permettant un verrou au niveau du flow du jeu grâce à une méthode "play()" impossible à surcharger, et des méthodes de décision déléguées au joueur.

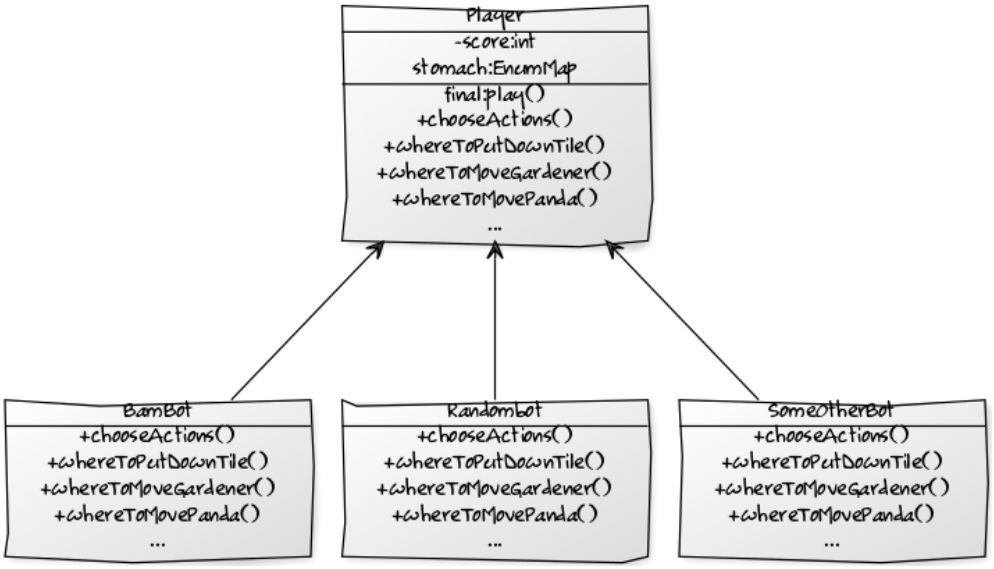


Image : diagramme simplifié de la structure des joueurs

Ainsi, un tour se déroule de la façon suivante : lors que le tour est donné à un joueur, il s'enquiert des actions que le joueur décide de faire. Le joueur devra anticiper à ce moment là les actions asynchrones telles que la dépose d'irrigation ou la validation d'objectifs, quitte à choisir de ne valider aucun objectif ou de garder son irrigation au moment de jouer son action. La classe Player lance ensuite la phase de jeu en elle même, après une rapide vérification de la légalité de la suite d'actions à jouer. A chaque décision (où bouger un pion, où poser une tuile ou une irrigation) elle donne la main au joueur, avant d'effectuer elle-même l'action sur le plateau.

Cette structure permet de n'avoir aucune autre contrainte sur le robot que de prendre les décisions. Lors du développement, il n'est pas nécessaire de penser à vérifier telle ou

telle possibilité, ou d'effectuer soi-même la pose des éléments ou les déplacements, puisque la classe Player s'en charge elle-même.

Pattern Factory

La pool d'objectifs

L'intérêt du pattern factory pour les objectifs s'est révélé dès les premières semaines et les premiers cours sur les design patterns. En effet il nous permet de ne pas avoir à instancier directement dans le jeu les objectifs, ce qui n'aurait pas permis la modularité la plus élémentaire : en effet rajouter ou supprimer un objectif des règles du jeu est une action simple qui ne devrait pas nécessiter d'adaptations majeures du déroulement des parties.

Dans le cas de nos objectifs, en plus de la gestion de l'instanciation des objets, d'autres problématiques ont apparu : en effet nous avons fait le choix de vérifier l'état de complétion des objectifs à chaque fois qu'un événement les affectant a lieu, plutôt que de le faire lorsque nous avons besoin de connaître cet état. Cela nous a conduit à devoir appeler des méthodes sur chaque objectif en jeu lors d'événements comme la pose de tuiles, la pousse d'un bambou, ou autre. Garder les objectifs ensemble dans une structure permettant d'appeler les méthodes de vérification de complétion sur chaque type en fonction des événements nous a semblé être une bonne solution pour rendre la gestion de cet aspect du jeu beaucoup plus lisible et transparent.

C'est de ces considérations qu'est né l'ObjectivePool, comme son nom l'indique un "pool" d'objectifs, qui gère leur instanciation, qui les stocke dans des listes spécifiques à leur type, et qui fournit des méthodes de notification des événements affectant la complétion des objectifs aux éléments du jeu.

Nous verrons plus loin que ce pool est un composant Spring dans notre version finale du projet, mais même avant cela, il suffit de changer le pool d'une partie pour avoir un jeu d'objectifs complètement différent, et ce de manière transparente.

Les robots

Nos robots étant des classes héritant de la classe Player, ils sont tous considéré dans le jeu comme des Players indifféremment de leur implémentation. Dès lors il semblait également naturel d'avoir recours à une factory pour les instancier, de manière à ce que faire concourir différentes répartitions d'intelligences artificielles les unes contre les autres ne nécessite pas de changer le code de notre moteur.

Nous avons mis cette factory en place assez tard, seulement au passage à Spring parce que nous n'avions pas plus de deux implémentations de Player et que nous avons estimé que le développement de nouvelles fonctionnalités était prioritaire jusqu'à ce moment. Par conséquent, la factory que nous avons mis en œuvre implémente une interface spécifique à Spring, et sera détaillée dans le paragraphe dédié à ce framework.

Pattern Singleton : pour un "properties file reader"

La classe "PropertiesFileReader" permet de récupérer des valeurs contenues dans des fichiers de propriétés telles que des valeurs de configuration, connexion, etc. Actuellement, elle est utilisée pour récupérer les constantes du jeu, telles que le nombre de

parties par tournoi, et c'est pourquoi le fichier de propriétés où sont ces constantes porte le nom de la classe "principale".

L'intérêt d'avoir un "properties file reader" consiste dans le fait d'avoir un découplage entre certaines valeurs, qui souvent sont utilisées à différents endroits de l'application, et le code : plutôt que de changer l'une de ces valeurs à plusieurs endroits du code, par exemple, on ne la change que dans le fichier des propriétés où elle est assignée. (Dans certaines architectures, on peut même avoir plusieurs fichiers des mêmes propriétés, qui varient en fonction de l'environnement par exemple : environnement de développement, par exemple, ou environnement de production, etc.)

Ainsi, l'accès à un "properties file reader" devait pouvoir se faire à partir de n'importe quel moment du code ; par ailleurs, l'accès à une donnée ne doit se faire qu'une fois dans le même temps. (C'est nécessaire pour l'écriture ; mais cela doit d'abord être le cas pour la lecture, car si une valeur, après avoir été récupérée, change et est récupérée une seconde fois (par une seconde instance de "reader") dans le même temps, par exemple, l'application peut se retrouver avec une valeur qui n'est pas la valeur attendue et avoir un comportement problématique.) Or les patrons de singleton nous assurent de n'avoir qu'une seule instance ; c'était donc ce qu'il nous fallait pour implémenter un "properties file reader".

Pour conclure sur ce point, nous précisons qu'il existe plusieurs manières d'implémenter une classe selon un patron de singleton et que nous avons choisi de concevoir la classe "PropertiesFileReader" de sorte qu'elle puisse répondre à une parallélisation de l'exécution de l'application : a priori, si l'application est lancée plusieurs fois en même temps (sur une même machine), l'instance de "PropertiesFileReader" restera unique.

Patrons envisagés puis rejetés

Nous avons à une étape de notre projet envisagé d'englober nos objectifs derrière une **façade**, mais la factory s'est révélée plus efficace. En effet, une façade permet de cacher du code et d'en rendre le pilotage extérieur plus simple, mais ne simplifie en rien le code en soi.

Pour les joueurs encore, nous avions envisagé le pattern **strategy**, mais le template s'est finalement imposé à cause de la simplicité que nous y avons trouvé pour intégrer de nouveaux robots.

L'architecture en termes de composants Spring

Comme demandé par le client, nous avons fait migrer notre projet dans le framework Spring. La migration s'est faite progressivement, et a été facilitée par le fait que nous avions conçu notre architecture en ayant à l'esprit l'importance du découplage entre les classes. Spring nous aura surtout permis d'accroître encore un peu plus ce découplage, et le plus gros travail que nous ayons eu à réaliser lors de sa mise en œuvre a été l'adaptation des tests.

Game et ObjectivePool

Nos deux premiers composants ont été la classe Game, qui représente le moteur de jeu, et la classe ObjectivePool, qui permet de faire l'interface entre le jeu et l'ensemble des objectifs. La classe Game ayant un champ ObjectivePool et cette dernière ayant également

un pointeur sur le Game en attribut afin de consulter l'état du jeu lors de la vérification de la complétion des objectifs, l'annotation Autowired n'a pas suffit et il a fallu dès le début faire appel au mécanisme du postconstruct, qui permets au Game d'envoyer sa propre référence à l'ObjectivePool à l'aide d'un setter. La classe principale a pu être adaptée avec un minimum de changements, nous avons utilisé l'interface ObjectFactory<> fournie par Spring, qui permet de créer automatiquement une factory de la classe en paramètre. Toutes les instances de la classe Game ont été créées par cette factory, et dès ce moment là nos composants principaux étaient instanciés automatiquement par spring, sans aucun appel au mot clé *new*.

La Board, les pions et les joueurs

Nous avons ensuite choisi de transformer en composants spring les éléments fondamentaux du squelette de notre architecture : la Board qui représente le plateau des tuiles posées, les pions Panda et Gardener, et finalement les différentes implémentations de Player.

Pour ce dernier composant, nous avons fait appel à l'interface FactoryBean<> qui nous permet de déléguer à la configuration spring le choix de l'implémentation de la factory de joueurs. Ces factories sont utilisées par les autres composants au moment où il y a besoin de créer des joueurs, permettant ainsi un découplage total entre le moteur de jeu et l'implémentation des joueurs. Les différentes factory n'ont même pas besoin d'être instanciées manuellement dans le code là où on en a besoin.

Ce qui aurait pu être un composant

Certains éléments auraient pu être représentés comme des composants Spring et ne l'ont pas été. Par exemple si notre logger n'avait pas été un attribut de la classe principale mais encapsulé dans une classe à part entière, il aurait été naturel d'en faire un composant singleton. Cependant étant donné la manière dont cet aspect a été implémenté dès le début et a évolué vers sa forme actuelle, qui est fonctionnelle, cela nous aurait demandé un travail que nous n'avons pas jugé prioritaire. Un autre élément est le properties file reader, en faire un composant spring aurait grandement simplifié son implémentation : en effet, il aurait suffit de l'annoter avec @Scope("singleton") pour que chaque instance automatiquement injectée soit la même, sans avoir à traiter cette problématique dans le code de la classe.

Ce qui n'est pas un composant

Pour finir, nous avons fait le choix de ne pas représenter certains aspects du jeu par des composants Spring, par exemple nos tuiles ou nos objectifs, qui sont des petits objets instanciés en grand nombre, ils ne font pas partie à proprement parler de l'architecture de notre application.

En conclusion

Notre implémentation nous permet une certaine modularité notamment apportée par l'utilisation d'interfaces et de classes abstraites pour représenter différentes parties du jeu, donnant à un possible autre développeur la possibilité de changer l'implémentation - par exemple - du plateau, ou encore de créer un nouveau robot ayant une stratégie de jeu différente.

Plus concrètement, implémenter l'extension *Chibis* du jeu ne nécessiterait pas un remodelage de fond en comble du code. L'exemple le plus flagrant étant l'addition de nouveaux objectifs qui ne demande qu'un ajout des cartes nécessaires dans l'ObjectivePool du jeu. De même, l'ajout des fonctionnalités du jeu de base non implémentées (ou pas complètement) telles que le dé météo peuvent l'être sans trop de difficulté.

D'autre part, la robustesse du programme est assurée par une forte couverture du code source par les tests unitaires (94.5% lors du rendu final).

Le jeu étant déjà en l'état intéressant en terme de stratégie et de mécaniques implémentées, nous aimerions concevoir des joueurs capables de prendre des décisions plus avancées que ceux existant actuellement, ce qui sera fait dans le cadre du travail d'étude et de recherche.