Projet Programmation 2
Deuxième Partie

Nicolas MARGULIES Stefan SCHWOON

4 mars 2025

1 Introduction

Pour la deuxième partie, le but est d'étendre la base codée en partie 1, notamment en diversifiant les comportements que différents objets d'un même type peuvent avoir, tout en utilisant le système de classes et l'héritage pour que cette complexité soit localisée dans le code desdites classes.

2 Le jeu à programmer

2.1 Organisation du code

Alors que votre projet va grandir en nombre et qualité des fonctionnalités, la taille de votre code va suivre. Il est donc crucial pour réduire le temps de développement et la probabilité de bugs que votre code reste le plus compact et organisé possible. Scala (entre autres langages) met à disposition la programmation orientée objet à ces fins, avec comme piliers l'héritage et l'encapsulation, dont vous devrez tirer parti. L'héritage permet un code compact, en réutilisant au maximum les parties en commun d'objets similaires. Ajouter un nouvel objet devrait se limiter à expliquer en quoi il diffère des types déjà existants. L'encapsulation (qui est plus une discipline qu'une fonctionnalité du langage) est le principe d'écrire au maximum le code concernant une classe dans la classe elle-même. Plus particulièrement, les modifications des données d'une classe devraient uniquement être faites par des méthodes de cette classe.

Ainsi l'ajout d'une fonctionalité peut se concentrer sur l'ajout d'une ou plusieurs classes et/ou méthodes, avec le moins possibles de modifications par ailleurs. Par exemple un produit inflammable ou périssable devrait hériter la plupart de ses fonctionnalités de la classe produit, et devrait gérer l'inflammation ou la péremption lui-même, avec le reste du code qui ne fait que l'informer du temps qui passe ou de la chaleur ambiante en appelant la méthode correspondante (implémentée sur tous les produits, mais qui ne ferait rien par défaut).

2.2 Déroulé du jeu

La principale différence avec la première partie est que dans la seconde partie, les machines doivent avoir un coût. Il y a principalement deux moyens de le faire : soit chaque machine a un coût en ressources, et le joueur possède une base dans laquelle on peut acheminer des ressources qui serviront à payer les machines, soit les machines doivent être fabriquées (via

d'autres machines) avant de pouvoir être placées (on peut même combiner les deux en devant fabriquer des machines, et les acheminer à la base avant de pouvoir les placer). Bien sûr, le joueur devrait démarrer avec suffisamment de ressources/machines pour avoir de quoi faire de nouvelles machines.

Ce mécanisme combiné à une plus grande diversité de ressources devrait conduire à une notion naturelle de progression dans le jeu : en fabriquant des objets de plus en plus complexes, on peut produire des machines de plus en plus complexes qui elles-mêmes nous permettent de produire des objets encore plus complexes, ect.

La diversité de ressources devrait venir avec plus qu'une grande liste de resources dont la seule différence est leur utilisation dans les recettes. Voici quelques indications (vous n'avez pas à toutes les implémenter ni à ne choisir que parmi cette liste) :

- On peut avoir certaines ressources qui sont des liquides, et qui ne se comportent donc pas de la même façon que les solides (transport par tuyaux, mesure en débit et non pas en temps de trajet, deux liquides ne peuvent pas être transportés dans le même tuyau, ect.).
- Certaines ressources peuvent réagir de manière spontanée si elles se trouvent sur la même case, soit créant un autre produit soit explosant.
- Certaines ressources peuvent être inflammables, et ne doivent donc pas être exposées à de trop fortes températures (et donc il faut avoir un moyen de calculer la température de chaque case en fonction des machines qui l'entourent).
- Certaines ressources peuvent être périssables, et soit se dégrader soit disparaître après un certain temps passé en dehors d'un contenant spécialisé.
- On peut avoir à gérer de l'énergie pour alimenter nos machines, voire plusieurs sources d'énergie (thermique, électrique, . . .) en fonction de la machine.

Une autre différence est que vous devez avoir des machines plus complexes. Ceci se présente selon trois axes :

- Des machines plus diverses : il devrait être possible de fabriquer des versions améliorées de certaines machines, ou des machines remplissant un rôle similaire mais différent (par exemple pour gérer d'autres types de ressources). Par exemple, un système de trains qui permet de transporter les ressources sur de grandes distances beaucoup plus rapidement que des convoyeurs, mais qui serait plus compliqué à mettre en place, et qui s'accompagnerait d'une discrétisation du transport (aka le train amène plein de ressources d'un coup, mais mettra du temps à repasser).
- Des machines programmables : en cliquant dessus un panneau permettant de configurer leur comportement devrait apparaître. Un exemple de base est que l'on devrait pouvoir changer l'orientation d'un convoyeur dynamiquement. L'exemple idiomatique de telles machines est une machine qui peut fabriquer toutes sortes d'objets selon un catalogue de recettes, et l'utilisateur doit choisir quelle recette la machine effectue (ce qui changera donc les ressources que la machine consomme et produit). D'autres exemples incluent des convoyeurs qui filtrent certaines ressources et les envoient dans une autre direction que le reste des ressources, ect.
- Des machines communiquantes : certaines machines doivent pouvoir envoyer et recevoir des signaux d'autres machines, et adapter leur comportement en fonction. Un exemple serait un moyen de compter la quantité de ressources dans la base pour automatiquement l'alimenter en ressources si besoin est, sans rediriger toutes les resources vers celui-ci. Ou un système générique de messages "j'ai besoin de X"/"je produit Y", qui permettrait à certaines machines de répartir automatiquement les ressources via un réseau de transport. Un tel système est très facilement Turing-complet, mais essayez de ne trop pas vous épargner de coder une fonctionnalité en disant que c'est au joueur de la coder.

Au vu de la nouvelle complexité du jeu, l'objectif peut être décalé de "être le plus efficace possible pour produire quelque chose de simple" à effectuer une action spécifique particulièrement

difficile (par exemple construire une machine spéciale qui requiert une grosse quantité de plein de ressources différentes). L'utilisateur devrait pouvoir choisir parmi différentes cartes qui peuvent varier de part leur taille, l'abondance de ressources présentes, voire des cartes à thème (manque ou surabondance d'une ressource, beaucoup de cases inaccessibles rendant le transport compliqué, des événements aléatoires pimentant le jeu, ect.)

2.3 Graphique

La partie graphique devrait un peu évoluer pour accomoder les nouvelles fonctionnalités (vous devrez probablement ajouter de l'affichage et des boutons). De plus, il faudra ajouter un minimum d'effets visuels pour que ce qui se passe à l'écran reflète bien ce qui se passe au niveau de la logique du jeu. Il faudra également veiller à inclure suffisamment d'informations à propos des boutons et des différentes machines pour que le jeu soit jouable sans avoir à regarder le code source.

Si vous vous en sentez l'envie, cela peut être l'occasion de développer les visuels de votre jeu. Cela peut inclure : remplacer les aplats de couleurs et les formes simples par des images, faire dépendre l'affichage d'un objet de son environnement (par exemple une case d'eau pourrait être différente en fonction de si elle est au milieu d'un lac, ou sur la rive), ou ajouter des animations.

Un autre axe d'amélioration est de pouvoir déplacer la vue actuelle de la carte afin de pouvoir gérer de manière correcte de grandes cartes.

3 Évaluation

3.1 Rapport et soutenance

Vous devez rendre un rapport de 2 à 3 pages et qui détaille vos choix techniques et les problèmes et difficultés que vous avez rencontrées. Dans le cas où certaines difficultés n'auraient pas pu être surmontées et que votre programme présenterait des défauts, vous pourrez expliquer ici ce que vous avez essayé d'entreprendre pour les résoudre et pourquoi cela n'a pas marché. Une soutenance de 15 minutes par groupe sera organisée à la fin de la première partie du projet où vous nous ferez une démonstration de votre programme.

3.2 Fonctionnalité du code

Votre projet sera évalué sur ses fonctionnalités. S'il remplit tout ce qui est demandé, rajouter d'autres fonctionnalités pourront apporter un bonus. La qualité graphique peut jouer un rôle, mais ce cours est avant tout un cours de programmation objet, ainsi, la perspective principale se portera sur les fonctionnalités, et l'évaluation de l'interface graphique reposera avant tout sur son caractère intuitif et facile à prendre en main.

3.3 Organisation du code

Votre projet devra être organisé de façon hiérarchique, et il vous faudra le séparer en fichiers, classes et méthodes. Il vous est recommandé de séparer le plus possible les différentes fonctionalités, notament les aspects *interface* (*frontend*) et *fonctionement du jeu* (*backend*).

3.4 Qualité du code

L'évaluation prendra en compte la qualité de votre usage de la programmation orientée objet ainsi que la qualité de votre utilisation du langage Scala. Faites attention à bien utiliser les propriétés d'héritages et à ne pas dupliquer du code inutilement. Utilisez plutôt des directives fonctionnelles que des boucles imbriquées. La mise en forme, la présence de commentaire et la cohérence des noms de classes, méthodes et variables devront être suffisamment décentes pour une lecture agréable du code.

4 Dates Importantes

— Deadline pour le rendu du code du projet : mardi 8 avril 18h
— Date de la soutenance de la première partie : jeudi 10 avril 11h15