Examen - HMIN102 ("Ingénierie Logicielle") - M1 Informatique

Christophe Dony - Clémentine Nebut

Année 2019-2020, Session 1, janvier 2020.

Durée : 2h00. Documents non autorisés. Il est possible de répondre aux questions de toute section sans avoir répondu aux autres ; mais la lecture dans l'ordre est vivement conseillée. Il y a 5 grandes sections rapportant respectivement environ : section 1-5 points, 2-4, 3-5, 4-4, 5-5. Une réponse excellente rapporte du bonus hors barème - et inversement une réponse au hasard ou très mauvaise vaut malus. La concision et le style des textes et programmes sont pris en compte.

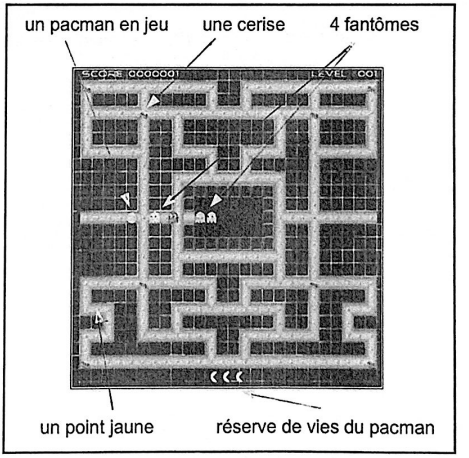
Contexte

Considérons comme contexte le jeu historique de type "Pacman". Les éléments d'une partie standard sont un héros, un *pacman*, et les éléments que le héros peut rencontrer que nous nommerons les "*obstacles*" ; les obstacles sont des *fantômes*, des *cerises* ou des *points jaunes* (voir figure). Les *points jaunes* et les *cerises* sont fixes. Les *fantômes* sont mobiles ; leurs déplacements sont aléatoires. Un *pacman* est mobile, son déplacement est contrôlé par le joueur (via le clavier par exemple).

Règles. (i)Un héros peut rencontrer lors de ses déplacements les autres éléments du jeu (des obstacles). (ii) Lorsqu'un *pacman* rencontre un *point jaune*, il le mange et son score augmente ; lorsque tous les points jaunes ont été mangés, le niveau courant est terminé et le jeu passe au niveau suivant. (iii) Lorsqu'un *pacman* rencontre un *fantôme*, il perd une vie ; s'il n'a plus de vies, la partie est terminée. (iv)Lorsqu'un *pacman* rencontre une *cerise*, il devient invincible par les fantômes pendant un certain laps (durée) de temps. Pendant ce laps de temps, une rencontre d'un *pacman* avec un *fantôme* envoie ce dernier en prison et rapporte des points supplémentaires ; à la fin du laps de temps, le pacman reprend son comportement standard.

Nous nous intéressons à la réalisation informatique de ce jeu, et par extension à la réalisation d'un framework extensible permettant l'intégration aisée de nouveaux types de héros (ou d'"obstacles") ayant des comportements différents lors des rencontres. A cet effet nous imaginons les classes ci-après (le décalage à droite signifiant "est sous-classe de" , par ex. `Obstacle` est sous-classe de `ElementJeu`). La détection des rencontres est implantée par le moteur du jeu (réalisé par une méthode moteur() de la classe `Jeu`). A chaque rencontre le moteur envoie un message `rencontrer(argument)` au héros courant (un *pacman* dans la version standard) avec comme argument l'objet rencontré, instance d'une sous-classe de la classe `Obstacle`.

```
Jeu
ElementJeu
  Héros
    Pacman
    AutreTypeDeHéros
  Obstacle
    Fantôme
        AutreTypeDeFantome
    Cerise
    PointJaune
    AutreTypeObstacle
```



1 Framework version No1, Bases de réutilisation

1. Dans une version No 1 très simplifiée de l'implantation du framework, on crée un nouveau jeu en créant une instance de la classe `Jeu` du listing 1 et en y injectant des éléments de jeu. Le moteur de jeu va surveiller les

```
1  public class Jeu {
2      protected Héros héros;
3      protected ArrayList<Obstacle> obstacles = new ArrayList<Obstacle>();
4      public void injectHeros(Héros h) { héros = h;}
5      public void injectObstacle(Obstacle o) {obstacles.add(o);}

7      public static void main(String[] args) {
8          Jeu g = new Jeu(); //un jeu avec un pacman et un fantôme
9          g.injectHeros( new Pacman() );
10         g.injectObstacle( new Fantôme() );}
11         g.moteurJeu()}
```

Listing 1 – classe Jeu de la version 1

rencontres entre le héros et les obstacles et invoquer une méthode `rencontrer(...)` correspondant au héros et à l'obstacle ; toute la question est de bien calibrer les définition et les envois de message afin (i) d'en invoquer une et (ii) d'invoquer la bonne.

a) Lors d'une exécution de la méthode `main` du listing 1, quelles sont les instructions qui entraînent directement ou indirectement l'exécution d'une affectation polymorphique ?Expliquez ?

b) Etablissez le lien entre affectation polymorphique et injection de dépendance dans le contexte d'un framework.

2. Dans cette première version du framework, on définit une méthode `rencontrer(Obstacle)` sur `Héros` comme dans le listing 2. Cette méthode s'adapte à tout nouveau type de `Héros` ou d'`Obstacle` grâce aux envois du message `description`.

```
1  public abstract class Héros extends ElementJeu {
2      public void rencontrer(Obstacle o){
3          System.out.println( this.toString() + " a rencontré un " + o.toString() + "!"); }

5      public String toString() { return ("Un héros"); } }
```

Listing 2 – classe `Héros` dans la version 1

a) Quelles sont les variables qui référencent les objets auxquels sont envoyés le message `toString()` dans la méthode `rencontrer(Obstacle)` ? Quel est le type statique de chacune de ces variables ?

b) Comment la méthode `rencontrer(Obstacle)` est-elle paramétrée ?

c) En considérant cette classe `Héros` comme une partie du Framework et les envois du message `toString()` comme des points d'extension, expliquer comment réaliser une extension (c'est à dire une nouvelle variante du jeu avec, par exemple, un nouveau type d'obstacle qui ferait voler un pacman capable de sauter vers une autre case). N'écrivez pas de code sauf si très court à titre d'exemple.

2 Framework version No2 - Spécialisation de méthodes

```
1  public abstract class Héros extends ElementJeu {
2      public void rencontrer(Obstacle r) {System.out.println(''Suis-je utile?'');}
3      public abstract void rencontrer(Fantôme f);
4      public abstract void rencontrer(Cerise c);
5      ... }

7  public class Pacman extends Héros {
8      public void rencontrer(Fantôme f){
9          System.out.println("pacman rencontre un fantôme"); }

11     public void rencontrer(Cerise c){
12         System.out.println("pacman rencontre une cerise"); } ... }
```

Listing 3 – classes `Héros` et Pacman de la version 2

```
1   Heros h = new Pacman();
2   Fantôme f = new Fantôme() ;
3   while (jeuNonFini){ //à chaque fois que le pacman rencontre un fantôme, on exécute :
4     h.rencontrer(f);
5     ...}
```

Listing 4 – Simulation du moteur de jeu pour la version 2

On souhaite maintenant implanter le fait que dans une vraie version du jeu, on doit exécuter un code différent lors de la rencontre entre un héros et des obstacles de différentes sortes. On oublie donc la version 1 précédente de la classe `Héros` et de la méthode `rencontrer(Obstacle)` et on propose la nouvelle version No 2 (voir 3 où est mise en place une solution pour tenter de distinguer les cas. Au niveau du moteur de jeu, on simule son comportement et l'envoi du message `rencontrer(...)` comme dans le listing 4. Prenez garde aux types des variables.

1. Pour réaliser la distinction des cas, pour quelle raison de génie logiciel a-t-on choisi ici de définir plusieurs méthodes `Rencontrer(...)` et pas une seule utilisant des tests via l'opérateur *instanceOf*?
2. Expliquez vos réponses aux deux questions suivantes en terme de : "envois de message", "type statique", "type dynamique" "redéfinitions de méthodes en présence de typage statique", etc. Il n'y a aucun code à écrire.
   a) Quelle est la méthode invoquée par l'appel `h.rencontrer(f)` du listing 4?
   b) Même question que a) en supposant que l'on n'ait pas défini la méthode "`public abstract void rencontrer(Fantôme f)`" sur la classe `Héros`.

3 Test du logiciel

Cette section porte sur le test du logiciel, nous allons travailler sur la position des éléments sur le plateau de jeu et pour cela ajouter de nouveaux attributs et méthodes sur les classes de l'application ; voir le listing 5.
- (i) On définit sur la classe `Jeu` un attribut `grille` représentant le plateau de jeu et ses chemins, initialisé à la création d'une partie par une matrice nxn de booléens ($n > 0$), avec `grille[i][j]` vaut vrai si la case de coordonnées (i, j) correspond à un chemin (en clair dans la figure) et faux si cette case correspond à un mur (en noir dans la figure). La méthode `placeEléments()` permet de placer chaque élément de jeu (héros et obstacles) dans la grille (donc de fixer ses coordonnées).
- On définit sur `ElementJeu`, les attributs `x` et `y` représentant les coordonnées de l'élément dans la grille, munis d'accesseurs en lecture et écriture, ainsi qu'une méthode `Jeu getJeu()` qui rend le jeu dans lequel se trouve l'élément.
- On définit sur la classe `Fantôme` une méthode `move()` qui déplace le receveur d'une case sur un chemin valide, c'est-à-dire ni dans un mur, ni en diagonale.

```
1   public class Jeu { //en complément aux sections précédents
2       ...
3       protected boolean[][] grille=new boolean[10][10];
4       public Jeu ( ArrayList<Obstacle> o) { ... remplissage des murs la grille ...}
5       public void placeEléments() { ... positionne les éléments de jeu dans la grille ... }
6       public int distanceBetween(ElementJeu e1, ElementJeu e2) { ... } ... }

8   public abstract class ElementJeu {
9       protected int x, y; // plus les accesseurs en lecture et écriture
10      ... }

12  public class Fantome extends Obstacle {
13      public void move() { ... algorithme de déplacement ...}
14      ...}

16  public class Pacman extends Heros {
17      public boolean estInvincible() { ... }
18      ... }
```

Listing 5 – Ajouts aux classes de l'application pour les questions sur le Test

1. On se place dans le cas (raisonnable) où il n'existe pas de case dans la grille qui ne soit pas un mur et qui soit

entièrement entourée de murs, et que donc un mouvement est toujours possible. Indépendamment de l'algorithme de déplacement utilisé pour le fantôme, comment proposez-vous de tester avec *JUnit* la propriété : *"suite à un appel à la méthode* `move()`*, un fantôme ne reste pas immobile (il a forcément bougé)"*? Décrivez succinctement les structures à mettre en place, et les instructions nécessaires au test de la propriété.

2. Quand le héros (un pacman ici pour simplifier) est invincible, l'algorithme de déplacement du fantôme vise à l'éloigner du pacman. On dispose dans la classe `Pacman` d'une méthode publique boolean `estInvincible()` qui retourne vrai quand le pacman est invincible. On dispose également dans la classe `Jeu` d'une méthode `int distanceBetween(ElementJeu e1, ElementJeu e2)` qui retourne la longueur du plus court chemin entre `e1` et `e2`.
   Comment proposez-vous de tester la propriété : *"quand dans le jeu il y a un pacman invincible, alors le déplacement d'un fantôme l'éloigne du pacman"*? Décrivez succinctement les structures à mettre en place, et les instructions nécessaires au test de la propriété.
3. Est-il possible (simplement) de s'assurer que lors du test effectué à la question précédente, la méthode `getX()` de pacman a été appelée au moins une fois? (même si c'est d'un intérêt nul dans notre cas). Justifiez.

4 Schémas de conception - Gérer les changements de comportement (pour la version No 2) du Framework

On revient dans le contexte de la version 2 du coeur du framework (listings 3 et 4) et on s'intéresse ici au problème indépendant de la réalisation du changement de comportement d'un *pacman*, déclenché lorsqu'il rencontre une cerise et faisant que, pendant un certain laps de temps et avant de redevenir normal, il envoie les fantômes qu'il rencontre en prison.

1. Décrivez en UML (précis, associations, déclarations attributs et méthodes) une solution logicielle utilisant le schéma de conception *State* pour implanter ce cahier des charges. Si vous souhaitez utiliser un autre schéma, précisez lequel et expliquez pourquoi il vous semble meilleur.
2. Donnez les éléments clé (n'écrivez que les parties du code en rapport avec la question posée) du code correspondant. L'aspect "comptage du temps écoulé" ne fait pas partie de la question et pourra être représenté par des commentaires dans le code. Précisez, où, quand et comment sont réalisés les changements d'état.

5 Framework version No 3, une version réutilisable réaliste du moteur de jeu.

La version 2 précédente du framework pose un problème pour l'écriture du moteur de jeu, il est en fait nécessaire de pouvoir stocker chaque nouvel objet rencontré dans une variable de type `Obstacle`. On garde les classes `Héros` et ses sous-classes inchangées par rapport à la version 2 et on réalise la simulation du moteur de jeu du listing 6.

```
1  Heros h = new Pacman();
2  while (jeuNonFini){
3     Obstacle r = //appel une méthode qui rend le prochain obstacle rencontré.
4     h.rencontrer(r);
5  }
```

Listing 6 – Simulation du moteur de jeu pour la version 3

1. En supposant que la variable `r` référence à l'exécution une instance de `Fantôme`, pourquoi l'appel `h.renconter(r)` invoque-t-il la méthode `rencontrer(...)` de la classe `Héros`?
2. Suite au problème indiqué par la question précédente, sans modifier la classe `Héros` ni ses sous-classes, proposez une modification du reste du système pour que l'ensemble fonctionne correctement avec toutes les sortes de héros et d'obstacles existants (invoquer la bonne méthode sur la bonne (sous-)classe de `Héros`). Il faut conserver cette variable `r` de type `Obstacle` dans le code du moteur de jeu (listing 6) mais vous pouvez modifier son code . Avez vous utilisé un schéma de programmation connu pour résoudre le problème? Si oui lequel?