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

Christophe Dony - Clémentine Nebut

Année 2019-2020, Session II, 16 juin 2020.

Durée : 1h20. L'énoncé est sur 4 pages. 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 4 grandes sections rapportant respectivement environ : section 1-6 points, 2-5, 3-4, 4-5. Une réponse excellente rapporte du bonus hors barème - et inversement une réponse au hasard, ou incohérente, ou dupliquée sans comprendre, ou auto-contradictoire, vaut malus. La concision et le style des textes et programmes sont pris en compte.

Contexte

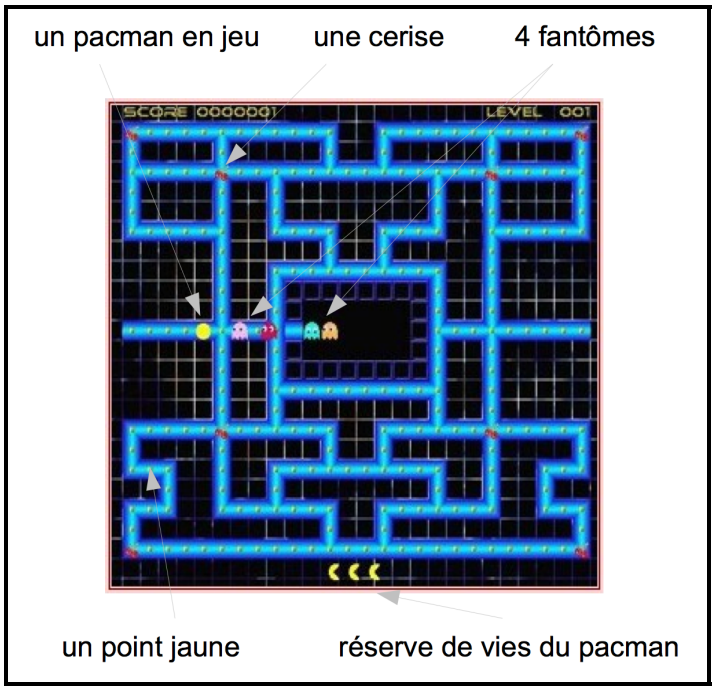
On reprend le contexte de l'examen de session I avec des questions différentes ou affinées. Lisez bien les codes, il peut y avoir des différences.

Considérons comme contexte le jeu d'arcade historique de type "Pacman". Les éléments d'une partie standard sont un héros [1], un *pacman*, et des éléments qu'un 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 plus globalement à la la réalisation d'un framework ou d'une ligne de produit intégrant du code extensible et réutilisable, permettant l'intégration aisée de nouveaux types de héros (ou de nouveaux types 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 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. Note : héros s'écrit avec un "s" en français et sans "s" en anglais.

```java
public class Jeu {
    protected Héros héros;
    protected ArrayList<Obstacle> obstacles = new ArrayList<Obstacle>();
    public void injectHeros(Héros h) { this.héros = h;}
    public void injectObstacle(Obstacle o) {this.obstacles.add(o);}

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

Listing 1 – classe `Jeu` de la version 1

1 Framework version No1, bases réutilisation et patterns

A) 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 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) Dans le listing 1, listez un exemple d'affectation polymorphique. Expliquez.

b) Pourquoi ne peut-on pas réaliser un framework sans utiliser d'affectation polymorphique ? Utilisez l'exemple du listing 1 pour expliquez votre réponse.

B) 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 à deux envois du message `getDescription()`.

```java
    public abstract class Héros extends Element {
        public abstract String getDescription();
        public void rencontrer(Obstacle o){
            System.out.println( this.getDescription() + " a rencontré un " +
                o.getDescription() + "!"); }
```

Listing 2 – classe `Héros` de la version 1

La méthode `rencontrer(Obstacle)` du listing 2 est paramétrée par spécialisation et par composition ; expliquez comment et ce que cela signifie du point de vue de l'utilisation du framework.

C) Dans une partie d'un jeu de *Pacman*, on souhaite qu'il ne puisse y avoir au maximum que 2 instances de la classe `Fantome`. Si une création d'instance supplémentaire est demandée, une exception spécifique `PlusDeFantomes` doit être signalée. Pour la classe `Fantôme`, donnez un code d'une variante du schéma *Singleton* permettant d'implanter cette spécification (donnez uniquement la partie du code correspondant à la question).

2 Framework version No2 - Spécialisation de méthodes, Configuration

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 numéro 2 (voir listing 3) où est mise en place une solution pour tenter de distinguer les cas. Au niveau du moteur de jeu, on place l'envoi du message `rencontrer(...)` comme dans le listing 4 (en l'étudiant, faites attention aux types statiques des variables).

A) Enoncez succintement mais précisément le problème de génie logiciel que pose l'utilisation de l'opérateur *instanceOf* dans un programme.

B) 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.

1) Quelle est la méthode invoquée par l'appel "`h.rencontrer(f)`" du listing 4 ? Expliquez votre réponse.

2) Même question que la "1)" ci-avant, en supposant que la ligne 2 du listing 4 soit remplacée par : "`Obstacle f = new Fantome();`". Expliquez votre réponse.

C) En considérant ce framework comme une ligne de produit, imaginez et présentez (à votre choix) un modèle de caractéristiques (*Feature Model*) permettant de générer des variantes du jeux.

```
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){
4        //à chaque fois que h rencontre un fantôme, on exécute :
5        h.rencontrer(f);
6         ...}
```

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

3 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.

A) Décrivez, en UML précis (n'oubliez pas les cardinalités, les attributs et méthodes utiles, une solution logicielle utilisant le schéma de conception **Decorator** appliqué à ce cahier des charges. On peut aussi gérer le problème avec *State*, mais *Decorator*, ici demandé, s'applique également puisqu'il permet de modifier un comportement pour un objet individuel.

B) Donnez les éléments clé (n'écrivez que les parties du code en rapport direct avec la question posée) du code correspondant. L'aspect "comptage du temps écoulé" n'est pas demandé et sera représenté par un commentaire dans le code. Précisez, où, quand et comment l'objet décoré remplace l'objet normal, et inversement comment l'objet redevient normal.

... Suite page suivante ...

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

La version 2 précédente du framework pose encore divers problèmes. Par exemple, pour l'écriture du moteur de jeu, il est nécessaire de pouvoir stocker chaque nouvel objet rencontré, quel que soit sa classe, dans une variable de type `Obstacle`, afin de pouvoir écrire l'instruction `h.rencontrer(o)` du listing 6. Pour ce faire, on modifie les classes `Héros` et `Pacman` selon le listing 5 et on réalise la simulation du moteur de jeu du listing 6.

```
1  public abstract class Héros extends Element {
2    public void rencontrer(Obstacle o) {o.rencontrer(this);}
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.print("Rencontrer(Fantôme) de la classe Pacman.");
10       //comportement d'un Pacman rencontrant un fantôme
11       ... }
12     ...}
```

Listing 5 – classes `Héros` et `Pacman` de la version 3

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

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

A) La méthode `rencontrer (Fantôme f)` de la classe `Pacman` est-elle une redéfinition de la méthode `rencontrer (Fantôme f)` de `Héros`, ou de la méthode `rencontrer (Obstacle f)` de `Héros`, ou des deux ? Expliquez.

B) Dans cette version, sans modifier le code de la version 3 (donc des listings 5 et 6) proposez, dans le contexte du schéma "double dispatch", une modification d'une ou de plusieurs des classes représentant les obstacles, pour que le framework fonctionne correctement sans utiliser aucun test "`instanceOf`". Par exemple, si la variable `o` du listing 6 référence à l'exécution une instance de `Fantôme` (respectivement `Cerise`), alors `h.rencontrer(o)` doit conduire finalement à l'exécution de la méthode `rencontrer (Fantôme f)` (respectivement `rencontrer (Cerise c)`) de la classe `Pacman`.

C) Montrez que la solution mise en place en **B)** fonctionne également, sans modification des méthodes existantes, et en ajoutant les méthodes nécessaires (donnez leur code), pour toute création a posteriori d'une nouvelle sorte d'obstacle. Si elle ne fonctionne pas, modifiez là.

D) Même question que **C)** mais pour une nouvelle sorte de héros.