TME 7 : Interface graphique

Objectifs pédagogiques : utilisation de SWING, boucle d'affichage, *Design Pattern Observer*

L'objet de ce TME est de permettre de visualiser le déplacement d'un agent dans le labyrinthe par l'utilisation du *Design Pattern Observer*. Le *DP Observer* définit une relation entre objets de type un-à-plusieurs de façon à ce que, lorsque un objet A (`Observable`) change d'état, tous les objets (`Observer`) qui se sont abonnés à la liste de diffusion de A soient notifiés et mis à jour automatiquement.

Dans notre cas, c'est la classe de simulation `Simulation` que l'on souhaite rendre `Observable`, et c'est la classe permettant de visualiser l'agent (une nouvelle classe `LabyViewer` qui reprend le code de `LabyBuilder`) qui jouera le rôle d'`Observer`. On fera donc dériver la classe `Simulation` d'une classe `SimpleObservable`, et l'interface graphique d'une interface `ISimpleObserver`. Ce schéma simple permet d'assurer la notification des `Observer` (les interfaces graphiques) sans créer de dépendance cyclique entre le modèle (ici la classe `Simulation`) et les interfaces.

⇒ Créez un projet « tme7 » et chargez-y l'archive de votre TME 6.

7.1 Mise en oeuvre du DP Observer

On souhaite permettre de visualiser le comportement d'un agent construit lors d'un des TME précédents. Pour cela, vous allez déployer une instance du Design Pattern *Observer* afin de synchroniser l'affichage et l'état de la simulation.

⇒ **Dans un `package pobj.obs`, construisez les classes et interfaces représentées en Figure 1.**

Ce schéma simple permet d'assurer la notification des `Observer` (typiquement les interfaces graphiques) sans créer de dépendance cyclique entre le modèle et les interfaces.

Les méthodes `addObserver(ISimpleObserver)` et `deleteObserver(ISimpleObserver)` servent respectivement à ajouter ou supprimer un `Observer` (dans un tableau ou dans un `ArrayList<ISimpleObserver>`). Les observers s'abonnent aux notifications de l'objet `Observable` en invoquant `addObserver`. Ils sont ensuite notifiés des changements d'état de l'objet Observable (appelé également « sujet ») à chaque fois que celui-ci invoque « `notifyObservers` ». Chaque `Observer`, dans le corps de sa méthode `update()`, peut alors décider comment traiter cette notification.

7.2 Mise à jour de la classe Simulation en la rendant observable

On souhaite pouvoir visualiser le comportement d'un agent dans le labyrinthe au cours d'une simulation. Il faut donc modifier la méthode `Simulation.mesurePerf()` afin de pouvoir accéder à l'état d'une simulation entre deux invocations de `agent.faitUnPas()` et rendre ainsi la simulation observable. Pour cela, procédez comme suit :

⇒ **Faites dériver `Simulation` de `SimpleObservable`. Vous invoquerez `notifyObservers()` après chaque invocation de `agent.faitUnPas`**, à l'intérieur de la boucle de la méthode `mesurePerf`.

Pour afficher les déplacements de l'agent à chaque pas, il faut pouvoir afficher l'état courant du labyrinthe, donc le contenu de chaque case. Pour cela, on interroge la simulation pour obtenir l'état du labyrinthe. Rappelons qu'une simulation modifie le labyrinthe sur lequel elle opère. Il faut donc parfois copier le labyrinthe avant de le lui passer.

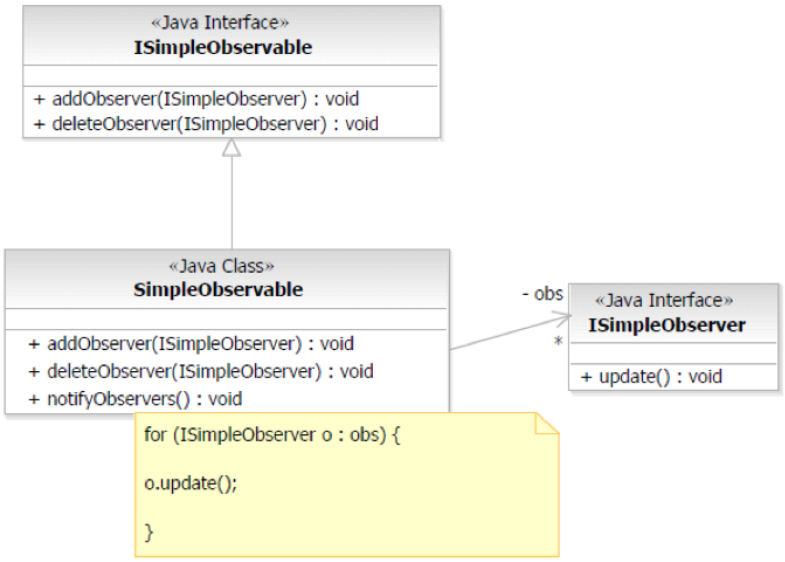

Figure 1: Diagramme UML des classes et interfaces pour le *Design Pattern Observer*.

7.3 LabyActivePanel : un observateur pour une Simulation

⇒ **Créez une nouvelle classe `LabyActivePanel`** qui dérive du `LabyPanel` et implémente `ISimpleObserver`. C'est cette classe que nous utiliserons pour dessiner le centre de la `JFrame` en lieu et place du `LabyPanel` de `LabyBuilder` (TME 6). Dans cette classe, **surchargez `actionPerformed`** de façon à ce que les *clics* sur les cases du labyrinthe n'aient plus d'effet (variante à faire si vous avez assez de temps : le clic sur une case doit positionner l'agent à cette position initiale).

⇒ **Implémentez** également **l'opération `update()`** héritée de `ISimpleObserver` en forçant l'affichage (dans la classe mère `LabyPanel`) à se rafraîchir.

NB: Pour ralentir l'affichage, vous pourrez ajouter un `sleep()` dans l'opération `update()` de mise à jour de votre `LabyActivePanel`.

```
try {
  Thread.sleep(500);
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

7.4 Ajout du conteneur `Swing LabyViewer`

⇒ Dans le package `agent.laby.interf`, en vous inspirant du code de la classe `LabyBuilder` fournie dans le package `agent.laby.interf`, **créez une classe `LabyViewer`** qui permet de visualiser le labyrinthe.

Le constructeur de `LabyViewer` prend un agent et un contrôleur et un nombre de pas en paramètre. Adaptez le constructeur récupéré de la classe `LabyBuilder` en fonction.

Pour assurer la mise à jour de l'affichage du labyrinthe, on va maintenant **modifier le `centerPanel`** de la `Frame` (cf. `CreateCenterPanel`() qui crée un `LabyPanel`) pour utiliser à la place un `LabyActivePanel`.

⇒ **Implémentez une méthode `main` dans cette classe**. Vous y intégrerez les traitements jusqu'à présent effectués dans vos méthodes `main` de test.

Vous récupérerez alors un des contrôleurs que vous avez créé au TME 6 et vous lancerez l'interface graphique `LabyViewer` afin de visualiser graphiquement le fonctionnement de ce controleur dans le

labyrinthe.

La classe `LabyViewer` servant uniquement à afficher le labyrinthe, veillez à supprimer les menus (cf. `createMenus()`) inutiles. De même, certains comportements n'ont pas d'intérêt ici (cf. `createSidePanel()`). Remplacez ceux-ci par un seul bouton «Play» permettant de lancer l'exécution de l'agent. Celui-ci doit déclencher la simulation, en invoquant `Simulation.mesurePerf(nbPas)`.

Chaque clic doit engendrer les actions suivantes :

1. copier le labyrinthe (méthode `clone()`),
2. créer une Simulation sur ce labyrinthe copié,

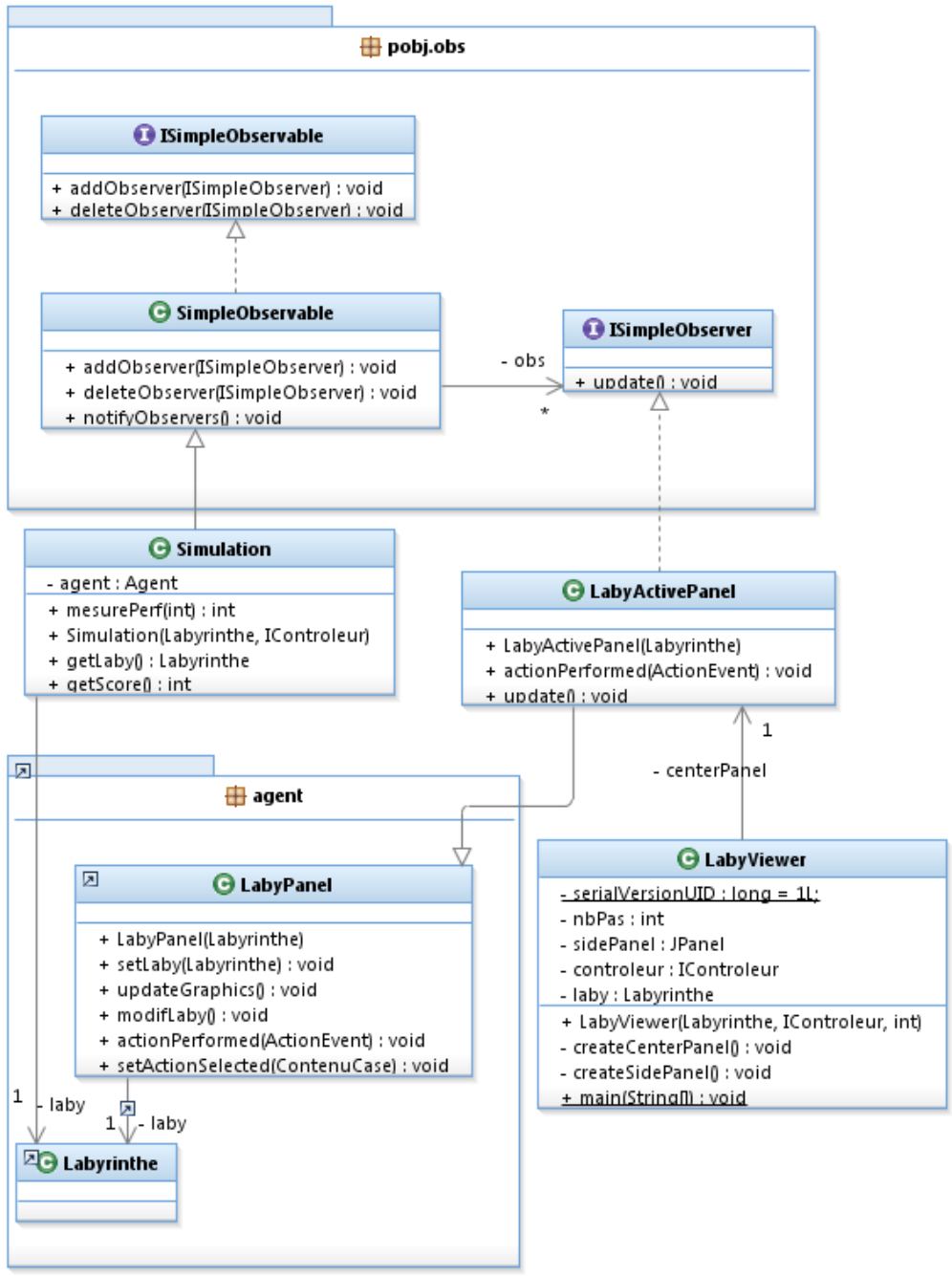

Figure 2: Diagramme UML pour l'intégration du DP Observer avec les classes utilisées pour la gestion de la simulation d'agents.

3. positionner le labyrinthe obtenu comme modèle du `LabyPanel` en invoquant `setLabyrinthe()`,
4. abonner le `LabyActivePanel centerPanel` à la simulation en invoquant `addObserver()`,
5. démarrer la simulation (methode `mesurePerf`),
6. désactivez l'abonnement de `centerPanel` grâce à la méthode `deleteObserver()`.

NB: En raison de la façon dont sont implémentés les contrôles Swing, il faut que le clic sur le bouton « play » soit terminé pour rafraîchir l'affichage. On peut invoquer une opération d'affichage dans un nouveau `Thread` et permettre ainsi de ne pas bloquer l'animation graphique. Utilisez telle quelle cette syntaxe anonyme un peu barbare pour lancer le traitement (`mesurePerf`) dans un nouveau `Thread` :

```
new Thread(new Runnable(){
      public void run() {
            // invoquer un traitement long ou une animation
      }
}).start();
```

⇒ Modifiez votre code en conséquence.

7.5 Améliorations

Ajoutez des contrôles permettant de choisir graphiquement (liste déroulante ou autre (`JTextField`, `ComboBox`...) le type de controleur à utiliser. Créez pour cela une nouvelle classe dérivée de `JPanel` et dédiée à la configuration du controleur. On pourra également ajouter un sélecteur permettant de changer de labyrinthe assisté par un FileChooser (à contrôleur constant).

7.6 Remise du TME

Envoyez un screenshot (jpg) de votre application finale