Autre exemple d'héritage avec la redéfinition

Revenons à un projet du chapitre 7 : le projet *zuul*. Dans le jeu du monde de Zuul, nous avons utilisé un ensemble d'objets `Room` pour créer la scène d'un jeu simple. L'un des exercices, vers la fin du chapitre, vous suggérait d'implanter une pièce transporteur (une pièce qui vous dirigeait vers un emplacement aléatoire du jeu si vous essayiez d'y entrer ou d'en sortir). Nous reprenons ici cet exercice, car la solution peut bénéficier grandement de l'héritage. Si vous avez des difficultés à vous remémorer ce projet, relisez rapidement le chapitre 7 ou examinez votre propre projet *zuul*.

Il n'existe pas de solution simple à ce problème. De nombreuses solutions sont possibles et peuvent fonctionner. Certaines sont cependant meilleures que d'autres. Elles peuvent être plus élégantes, plus faciles à lire, à maintenir ou à étendre.

Supposons que nous souhaitions implanter cette tâche de sorte que le joueur soit automatiquement dirigé vers une pièce aléatoire lorsqu'il essaye de quitter la pièce transporteur magique. La solution la plus simple et la plus évidente consiste à gérer cela avec la classe `Game`, qui implante les commandes du joueur. Une des commandes est « go », qui est implantée dans la méthode `goRoom`. Dans cette méthode, nous utilisons l'instruction suivante comme élément central de code :

```
nextRoom = currentRoom.getExit(direction);
```

Cette instruction récupère la pièce voisine de la pièce courante, dans la direction où nous souhaitons nous diriger. Pour ajouter notre transport magique, nous pourrions effectuer une modification comparable à celle-ci :

```
if(currentRoom.getName().equals("Pièce transporteur")) {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

Le principe est simple : nous vérifions si nous nous trouvons dans la pièce transporteur. Dans ce cas, la pièce suivante est obtenue de façon aléatoire (nous devons, bien sûr, implanter la méthode `getRandomRoom`). La suite du processus est la même que précédemment.

Bien que cette solution fonctionne, elle présente plusieurs inconvénients. Le premier est qu'il est préférable de ne pas utiliser des chaînes de texte, telles qu'un nom, pour identifier la pièce. Imaginez que quelqu'un veuille traduire votre jeu dans une autre langue — par exemple l'allemand. Il changera le nom des pièces — « Pièce transporteur » deviendra « Transporterraum » — et soudainement le jeu ne fonctionnera plus ! C'est un cas évident de problème de maintenance.

La seconde solution, qui est légèrement meilleure, consiste à utiliser une variable d'instance au lieu d'un nom pour identifier la pièce transporteur :

```
if(currentRoom == transporterRoom)  {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

Cette fois, nous partons du principe que nous avons une variable d'instance `transporterRoom` où nous stockons la référence de notre pièce transporteur[1]. À présent, le contrôle est indépendant du nom de la pièce. C'est un progrès.

Il reste encore à apporter quelques améliorations. Nous pouvons comprendre les faiblesses de cette solution lorsque nous envisageons une autre modification. Imaginons que nous souhaitions ajouter deux pièces transporteur supplémentaires.

Un aspect très séduisant de notre modèle était que nous pouvions en un seul point du code définir la structure du jeu, le reste du jeu étant totalement indépendant de cette organisation. Nous pouvions donc changer facilement la disposition des pièces, sans nuire au fonctionnement du reste — bon point pour la maintenance ! Avec notre solution actuelle, cependant, cela n'est plus du tout vrai. Si nous ajoutons deux nouvelles pièces transporteur, nous devons ajouter deux variables d'instance supplémentaires ou un tableau (pour stocker les références à ces pièces), et nous devons modifier notre méthode `goRoom` pour ajouter un contrôle pour ces pièces. En termes de facilité de maintenance, nous avons fait un pas en arrière.

La question est donc la suivante : pouvons-nous trouver une solution qui ne nécessite pas de modifier l'implantation de la commande à chaque nouvel ajout d'une pièce transporteur ? Voici notre proposition.

Nous pouvons ajouter une méthode `isTransporterRoom` dans la classe `Room`. De cette façon, l'objet `Game` n'a pas à se souvenir de toutes les pièces transporteur — les pièces le font elles-mêmes. Lors de la création de ces pièces, il serait possible de leur affecter un indicateur booléen précisant s'il s'agit de pièces transporteur ou non. La méthode `goRoom` pourrait alors utiliser le segment de code suivant :

```
if(currentRoom.isTransporterRoom()) {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

Nous pouvons maintenant ajouter autant de pièces transporteur que nous le souhaitons — aucune modification additionnelle de la classe `Game` n'est nécessaire.

1. Vous devez comprendre pourquoi un test d'égalité de référence est le plus approprié ici.

Cependant, la classe `Room` contient un champ supplémentaire dont l'utilité de la valeur ne tient qu'à la nature d'une ou deux instances. Du code lié à un cas particulier tel que celui-ci est typiquement révélateur d'une faiblesse dans la conception de classe. De plus, cette approche ne serait pas vraiment adaptable si nous décidions d'introduire d'autres types de pièces spéciales, chacune nécessitant son propre champ indicateur et méthode d'accès[1].

L'héritage nous permet de faire mieux et d'implanter une solution qui est encore plus souple que celle-ci.

Nous pouvons implanter une classe `TransporterRoom` comme sous-classe de la classe `Room`. Dans cette nouvelle classe, nous redéfinissons la méthode `getExit` et changeons son implantation de sorte qu'elle retourne une pièce aléatoire :

```
public class TransporterRoom extends Room
{

   /**
    * Retourne une pièce aléatoire, indépendante du paramètre
    * de direction.
    * @param direction Ignoré.
    * @return Une pièce aléatoire.
    */
   public Room getExit(String direction)
   {
      return findRandomRoom();
   }

   /*
    * Choix d'une pièce aléatoire.
    * @return Une pièce aléatoire.
    */
   private Room findRandomRoom()
   {
      ...// implantation non reproduite
   }
}
```

L'élégance de cette solution réside dans le fait qu'aucune modification n'est nécessaire dans les classes `Game` ou `Room` originales ! Nous pouvons simplement ajouter cette classe au jeu existant, et la méthode `goRoom` continue à fonctionner de la même façon qu'auparavant. Notez également que la nouvelle classe ne nécessite pas d'indicateur pour signaler sa nature particulière — son type et son comportement particulier fournissent cette information.

`transporterRoom` étant une sous-classe de `Room`, elle peut être utilisée à tout endroit où est attendu un objet `Room`. Elle peut donc être utilisée comme pièce voisine d'une autre pièce, ou être contenue dans l'objet `Game` comme pièce courante.

1. Nous pourrions aussi envisager d'utiliser `instanceof`, mais aucune de ces idées n'est optimale.

Nous avons omis, bien sûr, l'implantation de la méthode `findRandomRoom`. En réalité, il est préférable de la réaliser dans une classe séparée (disons `RoomRandomizer`) plutôt que dans la classe `TransporterRoom` elle-même. Nous laissons cela au lecteur en exercice.

Résumé

Lorsque nous avons affaire à des classes avec des sous-classes et des variables polymorphes, nous devons établir une distinction entre le type statique et le type dynamique d'une variable. Le type statique est le type déclaré, alors que le type dynamique est le type de l'objet actuellement stocké dans la variable.

Le contrôle de type est réalisé par le compilateur et utilise le type statique, alors que la recherche de méthode à l'exécution utilise le type dynamique. On crée ainsi des structures très souples par la redéfinition de méthodes. Même si l'on utilise une variable de supertype pour réaliser un appel de méthode, la redéfinition permet d'invoquer les méthodes spécialisées pour chaque sous-type particulier. De cette façon, les objets de classes distinctes peuvent réagir différemment au même appel de méthode.

Lors de l'implémentation de la redéfinition de méthodes, le mot clé `super` peut être utilisé pour invoquer la version superclasse de la méthode. Si des champs ou des méthodes sont déclarés avec le modificateur d'accès `protected`, seules les sous-classes de cette classe sont autorisées à y accéder.