Un exemple de réingénierie

Comme exemple, nous allons poursuivre l'extension de notre jeu concernant l'ajout d'objets. À la section « Cohésion des classes », nous avons suggéré d'utiliser une structure permettant de stocker un nombre quelconque d'objets dans une pièce. Il semble logique d'étendre notre programme afin qu'un joueur soit en mesure d'emporter les objets. Voici une spécification informelle de notre prochain objectif :

- Un joueur peut ramasser les objets de la pièce où il se trouve.
- Un joueur peut emporter un nombre donné d'objets, seul le poids total est limité.
- Certains objets ne peuvent pas être emportés.
- Un joueur peut déposer des objets dans la pièce courante.

Pour atteindre notre objectif, nous pouvons effectuer les tâches suivantes :

- Si ce n'est pas déjà fait, nous ajoutons une classe `Item` à notre projet. Comme nous l'avons déjà dit, un objet de type `Item` possède une description (une chaîne de caractères) et un poids (un entier).
- Nous devons aussi ajouter un champ `name` à la classe `Item`. Ce nom sera utilisé pour désigner un objet de façon plus concise qu'en utilisant la description. Par exemple, si un livre est présent dans la pièce courante, les valeurs des champs de cet objet pourraient être :

  name : `livre`

  description : un vieux livre poussiéreux en cuir gris

  weight : 1200

- Si nous entrons dans une pièce, nous pouvons afficher la description des objets présents pour en informer le joueur. Mais il sera plus commode d'utiliser le nom pour donner des commandes. Par exemple, le joueur pourrait saisir la commande `prendre livre` pour emporter le livre.

- Nous pouvons nous assurer que certains objets ne sont pas transportables en associant des poids très élevés (supérieurs à ce que peut emporter un joueur). Ou bien est-il préférable de définir un nouveau champ booléen `canBePickedUp` (« peut être emporté ») ? Quelle est la meilleure solution selon vous ? Ce choix est-il important ? Essayez de répondre à cette question en pensant aux évolutions possibles de ce jeu.

- Nous ajoutons les commandes `prendre` et `déposer` pour emporter et laisser sur place des objets. Les deux commandes sont composées à l'aide d'un nom d'objet comme deuxième mot.

- Nous devons définir quelque part un champ (désignant une collection) pour stocker les objets transportés par le joueur. Nous devons aussi ajouter un champ correspondant au poids maximal que peut transporter un joueur. Ce champ sera utilisé pour vérification quand le joueur souhaitera emporter quelque chose. Où devons-nous définir ces champs ? Réfléchissez de nouveau aux extensions futures pour prendre votre décision.

Nous allons maintenant détailler cette dernière tâche pour illustrer le principe de réingénierie.

La première question que nous devons nous poser quand nous décidons d'autoriser un joueur à emporter des objets est : où devons-nous définir les champs pour les objets transportés par le joueur et le poids maximal ? Un tour rapide des classes existantes montre qu'il n'est vraiment possible d'insérer convenablement ces champs que dans la classe `Game`. Ils ne peuvent pas être stockés par les classes `Room`, `Item` ou `Command`, car il existe de nombreuses instances de ces classes qui ne sont pas toujours accessibles au cours de l'exécution. Les classes `Parser` et `CommandWords` ne conviennent pas non plus.

Le fait que la classe `Game` stocke déjà la pièce courante (l'information concernant la position du joueur) renforce notre décision de placer les objets (l'information concernant les objets que possède le joueur) dans cette classe.

Nous pourrions programmer cette approche. Cependant, cette solution n'est pas bien conçue. La classe `Game` est déjà trop grosse et cela constitue un bon argument pour affirmer qu'elle contient déjà trop d'informations. En ajouter de nouvelles n'améliorera pas les choses.

Nous devons à nouveau nous interroger sur l'identité de la classe ou de l'objet qui devrait contenir ces informations. En réfléchissant au type d'information que nous ajoutons ici, nous nous rendons compte qu'elles concernent le *joueur* ! Il est donc logique (en suivant le principe de conception dirigée par les responsabilités) de créer une classe `Player`. Nous pouvons alors ajouter ces champs à la classe `Player` et créer un objet `Player` au début du jeu pour stocker ces données.

Le champ `currentRoom` stocke aussi des informations au sujet du joueur : la pièce où il se trouve. Par conséquent, nous devons aussi déplacer ce champ dans la classe `Player`.

Il est évident que cette conception respecte mieux le principe de conception dirigée par les responsabilités. Qui doit être chargé du stockage des informations concernant le joueur ? La classe `Player` bien sûr !

Dans la version originale, nous n'avions qu'une seule information concernant le joueur : sa localisation. Aurions-nous dû penser à une classe `Player` dès le début ? Cette question est à discuter. Nous pouvons avancer des arguments contradictoires. Argument en faveur de cette création : cela aurait été une bonne conception ; argument contre : définir une classe contenant un seul champ et aucune méthode peut être considéré comme exagéré.

Il existe parfois des zones d'ombre où les deux possibilités sont défendables. Mais la situation est plus claire après l'ajout des nouveaux champs. L'argumentaire en faveur d'une classe `Player` est maintenant fort. Cette classe stockera les champs et définira les méthodes telles que `dropItem` et `pickUpItem` (qui peut également vérifier la contrainte de poids et retourner la valeur `false` si nous ne pouvons pas l'emporter).

Ce que nous avons accompli en créant la classe `Player` et en déplaçant le champ `currentRoom` à partir de la classe `Game` vers la classe `Player` est une activité de réingénierie. Nous avons restructuré la façon dont l'information est représentée pour obtenir une meilleure conception en fonction de l'évolution des besoins.

Des programmeurs moins bien formés que nous (ou bien un peu plus paresseux) auraient laissé le champ `currentRoom` là où il était en remarquant que le programme fonctionnait ainsi et qu'ils ne ressentaient pas réellement le besoin d'effectuer cette modification. La conception de classe devient ainsi, peu à peu, désordonnée.

Nous pouvons illustrer les avantages de cette modification en réfléchissant à l'étape suivante. Supposons que nous souhaitions maintenant étendre le jeu pour permettre la gestion de plusieurs joueurs.

Grâce à notre nouvelle conception, cela devient tout à coup facile. Nous possédons déjà une classe `Player` (la classe `Game` contient un objet de type `Player`). Il est donc facile de créer plusieurs objets `Player` et de stocker une collection de joueurs dans la classe `Game` au lieu d'un seul. Chaque objet `Player` stockera sa propre pièce courante, ses objets et son poids maximal. Des joueurs différents pourraient même posséder des poids autorisés différents, ce qui étend le concept de joueurs multiples en autorisant des capacités différentes — cette capacité n'étant qu'une différence possible parmi d'autres.

Le programmeur paresseux qui a laissé le champ `currentRoom` dans la classe `Game` rencontre maintenant un problème sérieux : comme le jeu ne possède qu'une pièce courante, les positions des différents joueurs sont difficiles à stocker. Une mauvaise conception passe ainsi habituellement inaperçue initialement et finit tôt ou tard par être à l'origine d'un surcroît de travail.

Une bonne étude de réingénierie relève plus d'une réflexion guidée par un certain état d'esprit que de compétences techniques. Tout au long des modifications et extensions apportées à nos applications, nous devons nous interroger régulièrement et nous assurer que la conception actuelle des classes est toujours la meilleure solution. Au fur et à mesure que les fonctionnalités évoluent, les arguments en faveur ou en défaveur de certains choix de conception changent. Une conception qui était adaptée à une application simple peut ne plus convenir du tout après plusieurs extensions.

Au final, déceler ces changements et apporter les modifications de réingénierie nécessaires au code source économise du temps et épargne des efforts. Plus tôt nous faisons évoluer notre conception, plus nous réduisons le travail à accomplir.

Nous devons être préparés à scinder, réorganiser, des méthodes (extraire des groupes d'instructions d'une méthode pour en créer une nouvelle) et des classes (prendre des parties d'une classe pour en créer une nouvelle). S'interroger régulièrement sur l'opportunité de réingénierie permet de conserver une bonne conception de classes et d'économiser finalement du temps de travail. Bien entendu, la réingénierie nous compliquera la vie à long terme dans un cas : si nous ne la testons pas correctement par rapport à la version initiale. Dès que l'on entreprend une importante réingénierie, il est essentiel de vérifier au préalable que le test convient et qu'il est à jour tout au long de la procédure. Gardez cela en tête pour les exercices à venir.