TD 10 : Adapter, Composite, Decorator, Strategy

Objectifs pédagogiques : *Decorator*, *Composite*, *Stratégie*, *Adapter*

Ce sujet est basé sur une annale de LI314 2ème session, Janvier 2011. Il est fortement conseillé de lire l'énoncé de l'exercice en entier avant de vous lancer dans le codage.

10.1 Gestion d'énergie pour une batterie à bactéries à humus

En 2042, les batteries bactériologiques à base d'humus (HB pour « humus battery » en anglais) ont envahi le marché automobile.

Voici le fonctionnement d'une batterie « HB » :

- Une batterie dans sa globalité (ou « batterie globale ») est composée d'un ensemble de N cellules ou batteries « plus élémentaires ».
- La capacité d'une batterie est la somme des capacités des cellules qui la composent.
- Chaque cellule est elle-même une batterie (elle est chargeable et déchargeable) mais elle n'est pas décomposable (autrement dit, il n'existe pas de « sous-cellule »).
- Une batterie contient une certaine quantité d'énergie appelée sa charge, qui ne peut pas excéder sa capacité.

On se propose de modéliser une batterie « HB » en Java. Ainsi, une classe `Battery` représente une batterie « HB ». Une classe `Cell` représente une cellule.

10.1.1 Implantation des cellules et des batteries

Pour modéliser des cellules, la classe `Cell` contient :

- un constructeur paramétré par un entier indiquant sa capacité en unités d'énergie (un entier de type `int` fera très bien l'affaire). On fait l'hypothèse qu'à la construction, une cellule est complètement chargée.
- la méthode `public int getFullCapacity()` rendant la capacité potentielle de la cellule (c'est-à-dire quand elle est chargée au maximum) ;
- la méthode `public int getCurrentCapacity()` rendant la charge courante de la cellule ;
- la méthode `public boolean discharge()` renvoie `false` si la cellule était complètement déchargée ou décrémente la charge de 1 et renvoie `true` s'il restait de l'énergie à décharger.
- la méthode `public boolean charge()` incrémentant de 1 la charge de la cellule : `true` indique un succès, `false` indique un échec (la cellule était complètement chargée) ;

Pour modéliser des batteries, la classe Battery contient :

- un constructeur paramétré par deux entiers correspondant à un nombre de cellules et à leur capacité (on fait l'hypothèse que toutes les cellules d'une batterie ont la même capacité). On fait l'hypothèse qu'à la construction, une batterie est complètement chargée.
- la méthode `public int getFullCapacity()` rendant la capacité potentielle de la batterie (c'est-à-dire quand elle est chargée au maximum) ;
- la méthode `public int getCurrentCapacity()` rendant la capacité (charge/décharge) courante de la batterie ;
- des méthodes `public boolean discharge()` et `public boolean charge()` dont la sémantique, similaire à celle des cellules, sera précisée plus loin.

⇒ Quel design pattern pouvez-vous utiliser pour implanter les classes `Cell` et `Battery` ? Donnez le diagramme UML modélisant votre implémentation proposée.

On pourra se référer aux attributs des classes de ce diagramme dans le code demandé dans la suite.

⇒ Précisez le constructeur et le code des opérations `getCurrentCapacity()` et `getFullCapacity()` pour la classe `Battery`.

⇒ Donnez le code du constructeur et de `getCurrentCapacity()`, `getFullCapacity()`, `charge()` et `decharge()` pour la classe `Cell`.

10.1.2 Implantation du chargement et déchargement d'une batterie

Une batterie constituée de cellules n'a que peu d'intérêt si les cellules ne sont pas utilisées intelligemment. Deux approches de cette problématique sont envisageables :

- on peut recharger prioritairement la cellule dont la capacité courante est la plus faible, et décharger prioritairement la cellule dont la capacité courante est la plus forte ;
- ou bien on peut recharger et décharger les cellules en effectuant une ronde: on doit alors se souvenir de quelle cellule est la prochaine à charger et à décharger.

(NB: Oui, les deux approches ont plus ou moins le même effet global, mais peu importe pour la nature de l'exercice).

La charge et la décharge des batteries devront se faire avec des méthodes dont les signatures sont identiques à celles des cellules :

- la méthode `public boolean discharge()` renvoie `false` si la batterie était complètement déchargée ou décrémente la capacité de 1 et renvoie `true` s'il restait de l'énergie à décharger;
- La méthode `public boolean charge()` incrémente de 1 la capacité courante de la batterie : `true` indique un succès, `false` indique un échec (la batterie était complètement chargée).

⇒ Quel design pattern peut-on utiliser pour implanter les deux approches proposées ci-dessus ? Faites un diagramme UML de l'implémentation proposée.

⇒ Quel problème d'accès aux données se pose dans la définition du code des deux algorithmes ? A qui appartiennent les données qu'ils ont besoin de manipuler ? Comment leur transmettre cette information ? Quelle(s) opérations ajouter à `Battery` pour exposer le minimum requis pour pouvoir écrire les deux algorithmes ?

⇒ Donnez le code des constructeurs et des opérations de vos deux stratégies.

10.1.3 Batteries tuning

On souhaite équiper les batteries d'un dispositif anti-surcharge. En effet, charger une batterie qui a atteint sa capacité pourrait l'endommager.

⇒ Quel DP utiliser pour ajouter ce mécanisme anti-surcharge afin de protéger n'importe quelle batterie ? Faites un diagramme UML de votre solution.

Le *eBoost* est une technologie futuriste permettant de doubler la capacité des batteries qui en sont équipées. Ce système est vraiment génial: on peut consommer deux unités d'énergie pour une unité fournie en charge.

Il fonctionne de la façon suivante :

- Une décharge sur deux est ignorée sur une batterie équipée d'un eBoost (la deuxième initialement). Cependant si la batterie est vide, l'eBoost ne permet pas d'obtenir d'énergie.
- La charge se passe normalement, toute charge remet le système dans une position ou la prochaine décharge aura effectivement lieu.
- Les quantités `fullCapacity` et `currentCapacity` rendues par une batterie équipée d'eBoost sont celles de la batterie elle-même (les grandeurs ne sont pas doublées).

⇒ Proposez une façon d'implémenter ce mécanisme dans le même cadre.

10.1.4 Batteries rapides

Des bidouilleurs amateurs de courses de voitures ont parfois besoin de beaucoup de puissance sur une courte période pour une très grande accélération. Leur classe `Voiture` nécessite de pouvoir décharger les batteries de plusieurs unités à la fois en récupérant une unité par cellule non vide. Pour expliciter cela, ils ont défini une interface `IFastBattery` qui porte la méthode `public int discharge(int unitesDemandees)`. Elle prend en paramètres le nombre d'unités d'énergie qu'on souhaite récupérer et renvoie le nombre d'unités effectivement fournies.

Vous allez implanter ce nouveau type de batteries dans une classe `FastBattery`, qui s'appuie par délégation sur une batterie classique.

⇒ Quel design pattern peut-on utiliser pour réaliser la construction de ces batteries rapides ? Faites un diagramme UML de l'implémentation proposée.

⇒ Proposez une implantation de ce nouveau type de batteries dans une classe `FastBattery` dont vous fournirez le code (pensez à fournir un constructeur).