

Gabriel Bourgault - 1794069

Giuseppe La Barbera - 1799919

Remise du 6 décembre 2016

Réponses aux questions du TP5 - LOG2410

2. Patron Visiteur

1) **Intention** : Selon les notes de cours, l'intention du patron visiteur est de « représenter une opération qui doit être appliquée sur les éléments d'une structure d'objets. Un Visitor permet de définir une nouvelle opération sans modification aux classes des objets sur lesquels l'opération va agir. » En d'autres mots, cela permet d'ajouter une nouvelle fonctionné qui effectue des opérations sur des objets, tout en évitant d'avoir à modifier les dits objets. Dans le cas de PolyInfusion, on a donc les fonctionnalités [VisiteurCalculerPuissance](#) et [VisiteurCalculVolumeLiquide](#) qui sont ajoutées par le biais de visiteurs.

2) **Diagramme de classes** : Voir les documents PDF ci-joints :

[DiagrammeDeClasses_VisiteurCalculVolumeLiquide.pdf](#)

[DiagrammeDeClasses_VisiteurCalculerPuissance.pdf](#)

3) **Si en cours de conception vous constatez qu'il manque un type d'élément dans les circuits liquide (une sous-classe concrète de la classe ElmCircuitLiquide, par exemple la classe Valve), établissez la liste de toutes les classes qui doivent être modifiées.**

Tout d'abord, il faudrait nécessairement modifier les classes [MachineBase](#) ou [MachineLuxe](#) (dans le cosntructeur), afin d'ajouter le nouvel élément Valve. Il faudrait ensuite modifier les classes visiteurs : [VisiteurAbs](#), [VisiteurSansEffet](#), [VisiteurCalculerPuissance](#), [VisiteurCalculVolumeLiquide](#). Il faudrait donc leur ajouter des fonctions traiterValve(), par exemple. Finalement, et dépendant du comportement attendu de la machine, il faudrait peut-être modifier les classes [VisiteurTransfertLiqReservBouil](#) et [CommandeTransfertLiqReservBouil](#) afin qu'elles surchargent également traiterValve().

4) **Selon vous, le nettoyage de la machine pourrait-il être implémenté comme un visiteur ? Si oui, discuter des avantages et inconvénients d'utiliser le patron visiteur pour cette fonction et sinon expliquez pourquoi le patron n'est pas applicable.**

Il serait possible d'implémenter la fonctionnalité de nettoyage de la machine en tant que visiteur, mais cela pourrait s'avérer complexe à réorganiser. L'avantage principal d'utiliser un visiteur serait que la fonctionnalité n'aurait pas à être implémentée au sein des classes `MachineAbs`, `MachineBase` et `MachineLuxe`. Par contre, les machine de base et les machine de luxe n'ont pas le même processus de nettoyage (différentes composantes), ce qui introduit une grosse duplication lors de l'implémentation du visiteur, ce qui n'est pas souhaitable. Le template method utilisé dans les classes `MachineAbs`, `MachineBase` et `MachineLuxe` est à ce titre mieux adapté que l'usage d'un visiteur. Si l'on souhaitait tout de même utiliser un visiteur, il faudrait premièrement créer l'objet Visiteur désiré. Deuxièmement, il faut créer des méthodes afin de modifier les variables membres des classes affectées (chaque composante). Troisièmement, il faudrait déplacer le code des fonctions existantes (`nettoyer()`) vers le Visiteur créé.

3. Patron Commande

1 - a) **Intention** : Selon les notes de cours, l'intention du patron commande est d' « encapsuler une requête dans un objet de façon à permettre de supporter facilement plusieurs types de requêtes, de définir des queues de requêtes et de permettre des opérations « annuler ». » Donc, cela permet d'avoir une liste d'actions à effectuer.

1 - b) **Diagramme de classes** : Voir le document PDF ci-joint :

`DiagrammeDeClasses_Commande.pdf`

2) **Observez attentivement la classe `ExecuteurCommandes` qui permet de gérer la relation entre les commandes et les différents éléments de la théière. En plus de participer au patron Commande, elle participe à deux autres patrons de conception vu en cours.**

a) **Quel sont les noms et les intentions de ces patrons de conception ?**

- a. Singleton : « S'assurer qu'il ne soit possible de créer qu'une seule instance d'une classe, et fournir un point d'accès global à cette instance. » (Tiré des NDC). Bref, il faut que ce soit une classe qui est instanciée une seule fois et qui prévoit un mécanisme pour avoir accès à cette instance.

- b. Chain of responsibility : « Éviter de coupler l'émetteur et le récepteur d'une requête en donnant la possibilité à plus d'un objet de traiter la requête. Les objets récepteurs sont chaînés et la requête traverse la chaîne jusqu'à ce qu'elle soit traitée. » (Tiré des NDC).

b) Quels sont les éléments de la classe `ExecuteurCommandes` qui sont caractéristiques de ces patrons de conception ?

- a. On constate dans la classe `ExecuteurCommandes` qu'elle définit un seul chemin pour avoir accès à un objet de la classe (la fonction `getInstance()`). On voit aussi que le constructeur par défaut est privé, ce qui empêche d'instancier un objet de type `ExecuteurCommandes` sans passer par la méthode définie.

```
16  □ const ExecuteurCommandes* ExecuteurCommandes::getInstance()  
17      {  
18          if (m_instance == nullptr)  
19              m_instance = std::shared_ptr<ExecuteurCommandes>(new ExecuteurCommandes());  
20          return m_instance.get();  
21      }
```

- b. Les commandes sont déléguées à travers plusieurs classes jusqu'à ce qu'un objet soit en mesure de la traiter. Cela se rend en effet jusqu'au visiteur, qui est en mesure d'effectuer le traitement.

c) Pourquoi avoir utilisé ici ce patron de conception ?

En utilisant le patron Commande, on peut facilement effectuer différentes commandes (types de requêtes) de façon générique. Il suffit d'ajouter une commande dans l'`ExecuteurCommandes` qui les effectuera une à la suite de l'autre.

Le patron Singleton permet de s'assurer qu'il n'existe qu'une seule instance de l'`ExecuteurCommandes`. En encapsulant l'objet de cette façon, on obtient une plus grande flexibilité à long terme que de le définir en tant qu'objet statique.

Le patron Chain of Responsibility permet de déléguer la tâche à effectuer directement à la classe et à la fonction qui est en mesure de l'effectuer.

3) Pour compléter la fonctionnalité de la thèse, il faudrait ajouter de nouvelles sous-classes de la classe `CommandeAbs`. Selon vous, est-ce que d'autres classes doivent être modifiées pour ajouter les nouvelles commandes ? Justifiez votre réponse.

Si l'on voulait ajouter une nouvelle fonctionnalité à travers une commande, il ne suffirait que de créer cette nouvelle classe puis de l'ajouter dans le programme de commandes dans la `MachineBase`. En effet, le principe d'utiliser le patron commande est qu'il est simple d'en ajouter de nouvelles. Dans la nouvelle commande créée, il faut aussi ajouter un nouveau visiteur permettant d'accéder à la commande (ex : `CommandeNewSousClasse::VisiteurNewSousClass`). Par ailleurs, si cette nouvelle fonctionnalité requiert de nouvelles opérations qui ne sont pas permises par les objets que l'on vise, il faudra modifier ces derniers. Il pourrait s'agir, par exemple, d'un manque d'accès aux variables membres de la classe visée. Il serait aussi possible qu'il faille modifier les classes `VisiteurAbs` et `VisiteurSansEffet` (si l'on veut ajouter des méthodes) pour qu'elles soient disponibles à la classe `CommandeNewSousClasse::VisiteurNewSousClasse`.