HEIG-VD — POO

Laboratoire 7 (Hanoï) – Compte-rendu

[redacted]

20 octobre 2023

Ce document décrit nos choix et hypothèses de travail effectués pour réaliser le laboratoire n° 7 dans lequel nous avons dû créer un système permettant la résolution du problème des tours d'Hanoï avec un affichage par ligne de commande ou graphique. Ce laboratoire demande également la mise en place d'une pile établie à partir une liste simplement chaînée.

1 Question introductive

> "En supposant des moines surentraînés capables de déplacer un disque à la seconde, combien de temps reste-t-il avant que l'univers disparaisse (celui-ci a actuellement 13.8 milliards d'années) ?"

Dans l'antique légende hindoue, il est dit que la Tour de Brahmâ est composée de 64 disques d'or, situés sur la première des 3 aiguilles.
Afin de pouvoir déplacer l'intégralité des disques sur une autre aiguille, en estimant que les moines sont surentrainés et effectuent toujours les déplacements optimaux, ils devront effectuer $2^n - 1$ déplacements, soit $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ échanges.
Comme les moines déplacent un disque à la seconde, ils mettront donc

$$\frac{(2^{64} - 1)}{(365 \cdot 24 \cdot 3600)} = 5.849\,424\,173\,550\,720\,324\,\overline{391\,171\,99} \times 10^{11} \text{ années.}$$

Le déplacement complet de la pile mettra donc approximativement 584.94 milliards d'années.
Dans la légende, les disques sont déplacés depuis le commencement du monde, il resterait alors $584.94 - 13.8 = 571.14$ milliards d'années avant que le monde tombe en poussière et disparaisse.

2 Modélisation

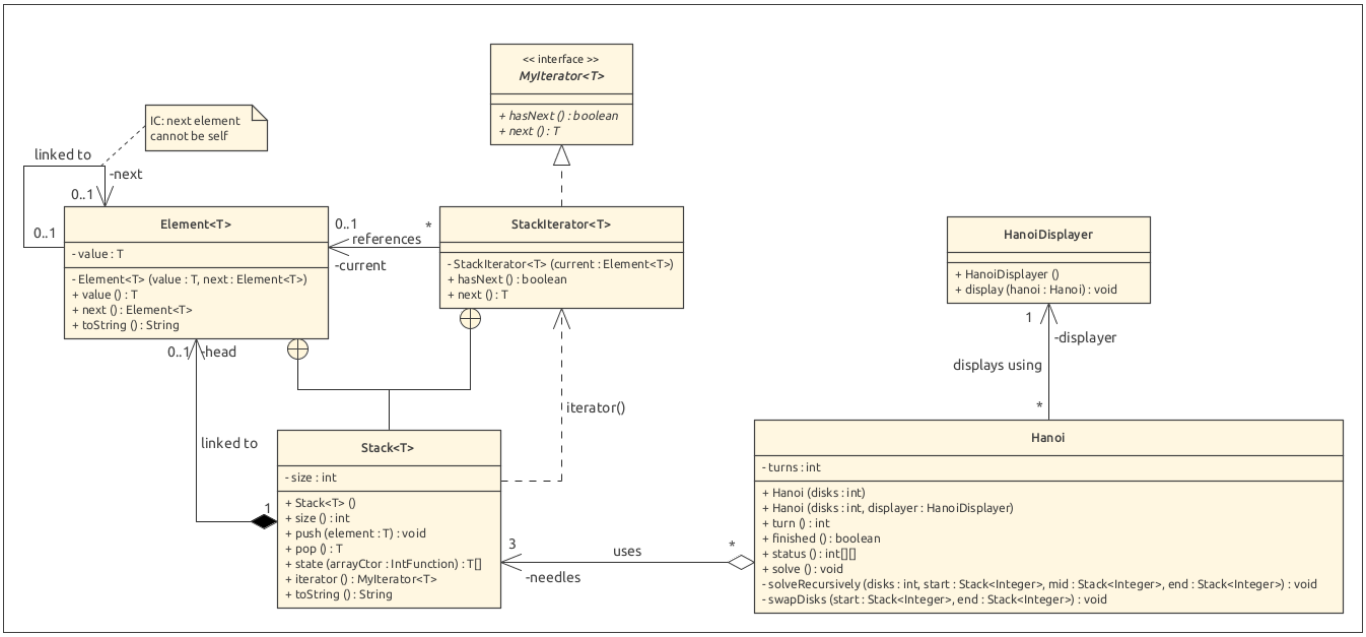

Figure 1 – Diagramme UML représentant les classes implémentées

3 Choix et hypothèses de travail

3.1 Paquetage `util`

Le paquetage `util` regroupe les classes utilitaires créées pour ce laboratoire et réutilisables dans d'autres projets. Afin de permettre une plus grande modularité de ces classes, elles ont été écrites de manière générique.

3.1.1 `Stack`

La classe `Stack` représente une pile générique, afin de pouvoir être réutilisée facilement dans d'autres projets. Elle comprend une classe `Element` ainsi qu'une classe `StackIterator`, décrites plus loin.
La classe `Stack` possède deux attributs, l'élément du sommet de la pile ainsi que la taille de la pile. La taille peut être obtenue grâce à l'accesseur associé. Concernant les valeurs de la pile, elles seront accessibles via la méthode `state()`, qui retourne le contenu de la pile sous forme de tableau. Les valeurs sont également visibles sous forme de chaîne de caractère avec la méthode `toString()`.
Étant donné la généricité de la classe, la méthode `state()` prend comme paramètre une référence vers un constructeur d'un tableau du type d'élément compris dans la pile. Cela permet d'éviter la création d'un tableau d'`Object` qui rendrait le paramètre générique peu utile.
Les interactions avec la classe `Stack` se feront essentiellement grâce aux méthodes `push()` et `pop()`, permettant respectivement d'ajouter ou de retirer l'élément du sommet. Il sera également possible d'itérer sur les différents éléments grâce à l'implémentation de l'interface `MyIterator`.

3.1.2 `Element`

La classe `Element` représente un élément de notre pile `Stack`. Pour cela, elle possède deux attributs, la valeur ainsi que l'`Element` suivant (si existant), tous deux avec leurs accesseurs respectifs. Ces deux attributs sont marqués `final`, car un élément ne changera pas durant son existence pour notre pile.
Cette classe est interne à la classe `Stack` pour permettre une implémentation privée qui limite la présence de classe tierces dans le paquetage `util`, sachant que son utilisation est unique et propre à `Stack`.

3.1.3 `MyIterator`

Cette interface est une définition plus ou moins semblable à celle fournie par `java.util`, en omettant la méthode `remove()` qui n'est pas prise en charge par `Stack`.
Cela permet d'exposer une API publique qui respecte l'encapsulation et limite au mieux des accès directs aux propriétés intrinsèques de l'implémentation propre à notre pile. Ainsi, l'utilisateur ne verra que deux méthodes lors de l'appel de `iterator()` : celles fournies dans cette interface.

3.1.4 `StackIterator`

Cette classe générique permet d'offrir une implémentation d'un itérateur sur la pile, proposant les opérations nécessaires pour la parcourir, à savoir `next()` et `hasNext()`. Afin de pouvoir mettre en place ces deux fonctions, la classe utilise un unique attribut de type `Element`, représentant l'élément sur lequel pointe actuellement l'itérateur. Cet attribut n'est pas visible de l'extérieur, car `Stack` expose son itérateur uniquement par le biais de l'interface parent `MyIterator`.
Cette classe est interne à `Stack` pour les mêmes raisons que `Element`, la classe étant privée cela offre à l'utilisateur aucune possibilité de facilement accéder aux champs inhérents à l'implémentation qui ne sont pas utiles dans un contexte d'itération.

3.2 Paquetage `hanoi`

Le paquetage `hanoi` regroupe les deux classes `Hanoi` et `HanoiDisplayer`, utilisées respectivement pour la résolution du problème des tours de Hanoï ainsi que pour son affichage en ligne de commandes.

3.2.1 `Hanoi`

La classe `Hanoi` représente donc un jeu de tours de Hanoï, de la situation initiale jusqu'à la résolution de ce problème bien connu.

3.2.1.1 Attributs

La classe `Hanoi` possède une constante `NUM_NEEDLES` afin de permettre une meilleure gestion des aiguilles ainsi que de laisser le choix de leur nombre, bien que cela demandera une possible révision de l'algorithme de résolution. Les aiguilles sont elles-mêmes représentées à l'aide d'un tableau de `NUM_NEEDLES` piles. Ce choix a été fait afin de permettre une meilleure gestion des aiguilles et de leur contenu, lors de l'affichage des états.
Le problème des tours de Hanoï conservant la même configuration du début à la fin de la partie, tous les attributs sont notés final, en dehors du nombre de mouvements effectués (attribut `turns`), variant bien évidemment au fil de la résolution.

3.2.1.2 Constructeurs

La classe `Hanoi` possède deux constructeurs, le premier permet uniquement de renseigner le nombre de disques utilisés et utilisera un affichage en ligne de commande. Le second constructeur permet de spécifier le type d'affichage à utiliser, dans notre cas la ligne de commande ou l'interface graphique fournie.
Lors de la construction de l'instance, la première aiguille sera automatiquement remplie avec le nombre de disques souhaité, avec la plus haute valeur représentant le disque le plus grand.
Le nombre de disques doit strictement être positif.
Le constructeur contient un `@SuppressWarnings("unchecked")` qui permet d'éviter certains warnings lors de la création du tableau de `Stack` générique (le tableau sera effectivement créé sous la forme d'`Object`). Il aurait probablement été préférable d'utiliser une collection style `ArrayList` qui n'aurait pas pénalisé la performance, et ferait la validation à notre place. Dans notre cas, le warning peut toutefois être ignoré.

3.2.1.3 Algorithme

Le problème des tours de Hanoï peut être résolu grâce à la méthode `solve()`. Celle-ci lancera l'affichage de l'état initial, avant d'appeler l'algorithme de résolution.
L'algorithme de résolution que nous avons choisi est celui qui est récursif. En effet, il privilégie une mise en place beaucoup plus simple que la version itérative, pour des performances en grande partie similaires.
On traite alors la solution comme suit : tant qu'il reste des disques à déplacer, on déplace les $n-1$ premiers disques sur l'aiguille intermédiaire, avant de déplacer le $n^{\text{ième}}$ disque sur l'aiguille de destination et d'y déplacer ensuite les $n-1$ disques restants.
De plus, dès qu'un disque est déplacé, on incrémente le numéro du tour actuel et l'on affiche l'état des aiguilles.

3.2.2 `HanoiDisplayer`

La classe `HanoiDisplayer` est utilisée pour l'affichage en ligne de commande. Pour cela, elle dispose d'une méthode `display` qui affichera le numéro du tour ainsi que la disposition des disques sur les différentes aiguilles.
Le seul attribut de cette classe est la constante `NEEDLE_NAMES` qui contient les noms associés aux différentes aiguilles. N'ayant donc rien à initialiser lors de la création des instances, le constructeur par défaut a été conservé.

4 Tests des cas limites

Avec la classe de test demandée, nous avons pu effectuer quelques vérifications des effets de bord concernant la classe `Hanoi`. Nous avons également créé une autre classe de test (`StackTest`) qui nous a permis de tester la classe `Stack` pour les scénarios ci-dessous.

Scénario	Résultat attendu
Accès à une pile vide	Les accesseurs affichent tous le résultat attendu
Pop d'une pile vide	`NoSuchElementException` levée dans la méthode
Push	Valeurs mises à jour et accesseurs fonctionnels
Push d'une valeur `null`	Valeur acceptée et accesseurs fonctionnels
Itérations de l'itérateur	`hasNext()` et `next()` fonctionnels
`next()` de l'itérateur complété	`NullPointerException` levée dans la méthode
Création de `Hanoi` avec un nombre de disques ≤ 0	`IllegalArgumentException` levée dans le constructeur

Table 1 – Tests effectués avec la classe `StackTest` et `Main`