HEIG-VD POO

# Rapport laboratoire n°7

Tours de Hanoï

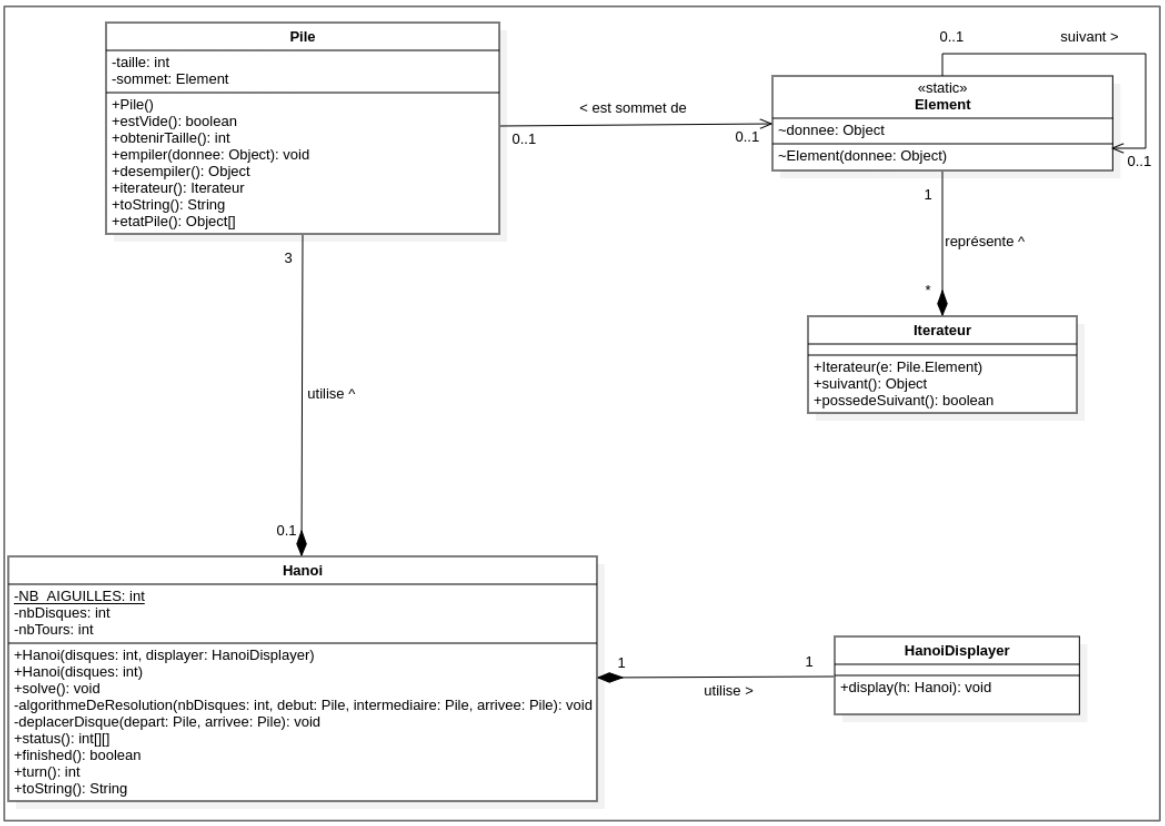
DECOPPET Joris, DUCOMMUN Hugo, MARTINS

12-9-2022

# 1 Introduction

Ce rapport concerne le laboratoire n°7 du cours de POO dans lequel nous avons dû implémenter le célèbre jeu des tours d'Hanoï. Le travail est séparé en deux parties majeurs. La première étant la création d'une structure de données de type « Pile » pouvant stocker des objets de types variés. La seconde partie consiste à ré-utiliser cette pile pour l'implémentation et la résolution de jeu. Il était aussi possible d'afficher le jeu de deux façons différentes, une étant via la console et la seconde via une interface graphique. L'utilisateur quant à lui pouvait choisir la taille de la tour.

# 2 Diagramme des classes



# 3 Description des classes

## 3.1 Structure de données

### 3.1.1 Elément

Cette classe a pour but de représenter un élément qui est utilisé dans la pile. Cet élément est composé d'une référence vers l'élément le suivant dans la structure, ainsi que d'une donnée pouvant être de type quelconque.

Cette classe a la particularité d'être en visibilité « Package » étant donné qu'elle n'a pas besoin d'être accédée par d'autres classes en dehors du package « Util ». Seules les entités liées à la pile ont besoin de pouvoir accéder à des éléments.

Cette classe est aussi une classe statique interne à la classe Pile. Cela améliore grandement la structure du code étant donné qu'un élément est utile uniquement dans le cas de la pile.

### 3.1.2 Itérateur

Un itérateur permet de référencé un élément. Il est notamment utile pour parcourir les éléments de la pile. Cet itérateur respecte les normes Java comme nous les avons vus en cours. Notamment pour la méthode « suivant » qui retourne la valeur de l'élément courant et passe au prochain élément.

Comme pour les éléments, cette classe est en visibilité « Package » pour les mêmes raisons.

### 3.1.3 Pile

Cette classe réunit les fonctionnalités des deux dernièrement présentées. Elle représente la structure de données en tant que telle. Elle permet d'avoir une pile fonctionnelle, de l'afficher et de la parcourir.

## 3.2 Jeu des tours d'Hanoï

### 3.2.1 Hanoï

Dans cette classe, on s'occupe du jeu lui-même. C'est-à-dire que nous avons nos trois aiguilles qui sont présentées sous la forme d'un tableau de trois piles. La classe implémente aussi l'algorithme de résolution du jeu qui sera détaillé dans la section suivante. L'affichage du jeu quant à lui se passe dans une classe séparée.

### 3.2.2 HanoiDisplayer

Cette dernière classe nous permet de faire le lien entre le jeu et son affichage par l'utilisateur. Elle permet d'appeler les méthodes nécessaires présentent dans Hanoi pour afficher le jeu. La version graphique du jeu quant à elle est gérée par une libraire externe (JHanoi) qui nous était fournie par les enseignants. Nous n'avons fait que de l'utiliser en lui fournissant le jeu à afficher et sa résolution.

# 4 Description de l'algorithme

Pour la réalisation de l'algorithme de résolution du jeu, nous avons décidé de partir sur sa version récursive. Nous étions déjà familiers avec celui-ci grâce au cours d'ASD et il est surtout bien plus court et facile à implémenter. Nommons les aiguilles de 1 à 3, respectivement de gauche à droite. Et utilisons un jeu composé de « N » disques où N est un entier strictement positif.

- Déplacer N-1 disques de la première tour à la seconde. Ce mécanisme se réalise grâce des appels récursifs.
- Il ne reste alors que le plus grand des disques non-placé sur la première aiguille. On déplace celui-ci sur la troisième aiguille.

- On déplace les N-1 premiers disques de la seconde à la troisième aiguille.

Cela nous demande alors de réaliser $2^N - 1$ déplacements pour résoudre ce problème.

# 5 Liste des tests réalisés

Tous les tests cités ci-dessous ont été concluants. Le résultat attendu était à chaque fois le résultat obtenu.

| Description du test | Résultat attendu |
|---|---|
| **Itérateur** | |
| Itérer sur des éléments d'une pile | Les éléments sont itérés dans l'ordre et dans leur totalité |
| Itérer sur une pile vide provoque une exception | Exception levée |
| Un itérateur possédant un élément suivant indique que c'est le cas | L'itérateur indique bien qu'un élément suivant est présent |
| **Pile** | |
| Une pile vide est indiquée vide | La pile est vide, alors « vrai » est retourné |
| Taille de la pile annoncée correctement | La pile a la taille espérée |
| Ajouter des éléments sur la pile et les désempiler pour vérifier qu'ils sortent dans le bon ordre. | Les éléments empilés dans l'ordre<br>A -> B -> C ressortent dans l'ordre<br>C -> B -> A |
| Désempiler une pile vide | Une exception est levée |
| Affichage de la pile | Une pile remplie avec les éléments<br>A -> B -> C s'affiche de la manière<br>[ <C> <B> <A> ] |
| L'état de la pile est correctement réalisé dans un tableau | Une pile remplie avec les éléments<br>A -> B -> C nous donne le tableau suivant<br>[0] -> C, [1] -> B, [2] -> A |
| **Hanoi** | |
| Résolution du jeu correcte | A la fin du jeu les disques sont dans le bon ordre sur la dernière aiguille |
| Jeu terminé | Lorsque le jeu est terminé, la méthode corresponde nous indique que c'est le cas |
| Jeu en cours | Tant que le jeu n'est pas fini, la méthode nous indique qu'il est en cours |
| Affichage du statut du jeu | Le jeu nous est donné sous le format d'un tableau bidimensionnel- |

# 6 Réponse à la question

Comme expliqué lors de la description de notre algorithme. Le nombre de mouvements nécessaires pour compléter un problème de N disques est de $2^N - 1$. Dans le cas où les moines doivent déplacer 64 disques, on a alors $2^{64} - 1$ mouvements et étant donné qu'ils font un mouvement par seconde, on a alors ce même nombre de secondes.

La conversion de ce très grand nombre en milliards d'années nous donne alors précisément 584.9 milliards d'années. Si on y soustrait l'âge de l'univers, on obtient alors **571.2 milliards d'années** restants avant la fin de l'univers. Tous ces calculs ont été réalisés grâce à WolframAlpha.

```java
package app;

import hanoi.*;
import hanoi.gui.*;

/**
 * Classe principale du programme
 * @author Decoppet Joris, Ducommun Hugo, Martins Alexis
 */
public class App {

    /**
     * Méthode main de la classe principale
     * @param args Arguments du programme pour passer les nombres de
     * disques
     */
    public static void main(String[] args) {
        /*
         * Si lors de l'appel de notre classe, un argument est passé, alors
         * c'est que l'on souhaite afficher le jeu au format console.
         */
        if (args.length > 0) {
            int nbDisques = lireArgument(args);
            Hanoi hanoi = new Hanoi(nbDisques);
            hanoi.solve();
        } else {
            new JHanoi();
        }
    }

    /**
     * Verifier qu'il y ait bien un seul argument et qu'il soit un entier
     * @param args les arguments de la ligne de commande
     * @throws IllegalArgumentException Si on n'a pas un seul argument ou
     * si l'argument n'est pas un entier
     */
    private static int lireArgument(String[] args) {
        if (args.length > 1) {
            throw new IllegalArgumentException("Il faut au plus 1 argument. " +
                      "0 arguments pour l'affichage graphique, " +
                      "1 argument pour l'affichage console");
        }
        try {
            return Integer.parseInt(args[0]);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Le nombre de disques " +
                                               "doit être un entier");
        }
    }
}
```

```java
package hanoi;

import util.Pile;

/**
 * Cette classe représente le jeu des tours d'Hanoi. Elle modélise les trois
 * aiguilles du jeu, ainsi que les disques qui sont dessus.
 * Elle permet aussi de résoudre le problème et d'afficher les résultats
 * via un affichage console ou graphique de la classe HanoiDisplayer.
 * @author Decoppet Joris, Ducommun Hugo, Martins Alexis
 */
public class Hanoi {
    // Nombres d'aiguilles du jeu d'Hanoi
    private static final int NB_AIGUILLES = 3;
    // Nombre de disques du jeu
    private final int nbDisques;
    // Nombre de tours réalisés à un instant t
    private int nbTours;
    // Représentation du jeu
    private final Pile[] aiguilles;
    // Afficheur du jeu
    private final HanoiDisplayer displayer;

    /**
     * Constructeur de la classe Hanoi.
     * @param disques le nombre de disques à utiliser
     * @param displayer l'afficheur à utiliser
     * @throws IllegalArgumentException Si le nombre de disques est inférieur à 1
     */
    public Hanoi(int disques, HanoiDisplayer displayer) {

        if(disques < 1) {
            throw new IllegalArgumentException("Le nombre de disques doit être " +
                                               "un entier positif");
        }

        this.nbDisques = disques;
        this.nbTours = 0;
        this.displayer = displayer;
        this.aiguilles = new Pile[NB_AIGUILLES];

        // Initialisation des aiguilles
        for (int i = 0; i < NB_AIGUILLES; ++i) {
            this.aiguilles[i] = new Pile();
        }

        // On empile les disques sur la première aiguille
        for (int i = this.nbDisques; i > 0; i--) {
            this.aiguilles[0].empiler(i);
        }
    }

    /**
     * Constructeurprenant uniquement les disques en paramètre.
     * @param disques Nombre de disques
     */
    public Hanoi(int disques) {

        this(disques, new HanoiDisplayer());
    }

    /**
     * Résoud le problème des tours d'Hanoi Appel à la fonction récursive.
     */
    public void solve() {

        this.displayer.display(this);
        algorithmeDeResolution(this.nbDisques, this.aiguilles[0],
                               this.aiguilles[1], this.aiguilles[2]);
    }

    /**
```

```
     * Algorithme récursif de résolution du problème des tours d'Hanoi.
     * @param nbDisques Nombre de disques sur l'aiguille
     * @param depart Aiguille de départ
     * @param intermediaire Aiguille intermédiaire
     * @param arrivee Aiguille d'arrivée
     */
    private void algorithmeDeResolution(int nbDisques, Pile depart,
                                        Pile intermediaire, Pile arrivee){
        // Cas trivial dans lequel on déplace juste 1 disque
        if (nbDisques == 1) {
            deplacerDisque(depart, arrivee);
        } else {
            // Algorithme d'Hanoi récursif
            algorithmeDeResolution(nbDisques - 1, depart, arrivee, intermediaire);
            deplacerDisque(depart, arrivee);
            algorithmeDeResolution(nbDisques - 1, intermediaire, depart, arrivee);
        }
    }

    /**
     * Déplace un disque d'une aiguille à une autre et incrémente
     * le nombre de tours.
     * @param depart Aiguille de départ
     * @param arrivee Aiguille d'arrivée
     */
    private void deplacerDisque(Pile depart, Pile arrivee) {
        arrivee.empiler(depart.desempiler());
        ++nbTours;
        this.displayer.display(this);
    }

    /**
     * Retourne l'état de jeu d'Hanoi sous forme de tableau bidimensionnel.
     * @return Tableau bidimensionnel représentant l'état du jeu.
     */
    public int[][] status() {

        int[][] ret = new int[this.aiguilles.length][];

        for (int i = 0; i < NB_AIGUILLES; ++i) {
            // Récuoération du tableau 1D d'une aiguille
            Object[] etat = this.aiguilles[i].etatPile();

            // Création d'un tableau de la taille de l'aiguille
            ret[i] = new int[etat.length];

            // Affectation du tableau case par case
            for (int j = 0; j < etat.length; ++j) {
                ret[i][j] = (int) etat[j];
            }
        }

        return ret;
    }

    /**
     * Indique si le jeu s'est terminé en vérifiant que tous les disques
     * nécessaires sont sur la dernière aiguille.
     * @return  Vrai --> Jeu fini
     *          Faux --> Jeu non fini
     */
    public boolean finished() {
        return this.aiguilles[NB_AIGUILLES - 1].obtenirTaille() == this.nbDisques;
    }

    /**
     * Retourne le nombre de tours effectuées
     * @return Nombre de tours effectuées
     */
    public int turn() {

        return this.nbTours;
```

```
    }

    public String toString() {

        // En-têtes des aiguilles lors de l'affichage
        final String[] enTetes = {"One:", "Two:", "Three:" };
        StringBuilder sb = new StringBuilder();

        sb.append("-- Turn: ");
        sb.append(this.nbTours);
        sb.append("\n");

        // Affichage des piles représentant l'aiguille
        for(int i = 0; i < NB_AIGUILLES; ++i) {
            sb.append(String.format("%-7s", enTetes[i]));
            sb.append(this.aiguilles[i]);
            sb.append("\n");
        }

        return sb.toString();
    }
}
```

```java
package hanoi;

/**
 * Cette classe permet d'afficher le jeu des tours d'Hanoi pour la version console.
 * @author Decoppet Joris, Ducommun Hugo, Martins Alexis
 */
public class  HanoiDisplayer {

    /**
     * Affiche le jeu des tours d'Hanoi en console.
     * @param h le jeu des tours d'Hanoi à afficher
     */
    public void display(Hanoi h) {
        System.out.println(h);
    }
}
```

```java
package util;

import java.util.*;

/**
 * Cette classe permet de modéliser la structure de données Pile.
 * Elle possède une classe interne modélisant ses éléments.
 * @author Decoppet Joris, Ducommun Hugo, Martins Alexis
 */
public class Pile {

    /**
     * Cette classe représente un élément de la pile.
     * @author Decoppet Joris, Ducommun Hugo, Martins Alexis
     */
    static class Element {
        // Donnée stockée dans l'élément
        final Object donnee;
        // Elément suivant dans la pile
        Element suivant;

        /**
         * Constructeur de la classe Element.
         * @param donnee Donnée à stocker dans l'élément
         */
        Element(Object donnee) {
            this.donnee = donnee;
            this.suivant = null;
        }
    }
    // Elément au sommet de la pile
    private Element sommet;
    // Taille de la pile
    private int taille;

    /**
     * Constructeur de la classe Pile.
     */
    public Pile() {

        this.taille = 0;
    }

    /**
     * Indique si la pile est vide
     * @return Vrai --> La pile est vide
     *         Faux --> La pile n'est pas vide
     */
    public boolean estVide() {

        return this.taille == 0;
    }

    /**
     * Retourne la taille de la pile
     * @return taille de la pile
     */
    public int obtenirTaille() {

        return this.taille;
    }

    /**
     * Rajoute un élément au sommet de la pile
     * @param donnee Elément à rajouter
     */
    public void empiler(Object donnee) {

        Element nouveauPremier = new Element (donnee);
        /*
         * Ajout du nouveau sommet de la pile et le précédent sommet est
         * identifié comme le suivant du sommet actuel
```

```
         */
        nouveauPremier.suivant = this.sommet;
        this.sommet = nouveauPremier;
        ++this.taille;
    }

    /**
     * Enlève l'élément qui est au sommet de la pile et retourne sa valeur.
     * @return Valeur de l'élément supprimé
     * @throws NoSuchElementException Génère une exception si la pile est vide.
     */
    public Object desempiler() {

        // On ne peut pas désempiler s'il n'y a rien sur la pile
        if (this.estVide())
            throw new NoSuchElementException();


        Object donnee = this.sommet.donnee;
        this.sommet = this.sommet.suivant;
        --this.taille;
        return donnee;
    }

    /**
     * Création d'un itérateur sur le sommet de la pile
     * @return Itérateur sur le sommet de la pile.
     */
    public Iterateur iterateur() {
        return new Iterateur(sommet);
    }

    /**
     * Retourne la représentation textuelle de la pile.
     * @return Chaîne de caractères représentant la pile.
     */
    public String toString() {
        StringBuilder sb = new StringBuilder();
        Iterateur iterateur = iterateur();

        sb.append("[");

        while(iterateur.possedeSuivant()) {
            sb.append(" ");
            sb.append("<");
            sb.append(iterateur.suivant());
            sb.append(">");
        }

        return sb.append(" ]").toString();
    }

    /**
     * Tableau unidimensionnel qui représente l'état de la pile avec ses éléments.
     * @return Tableau unidimensionnel représentant l'état de la pile.
     */
    public Object[] etatPile() {

        Object[] etat = new Object[taille];
        Iterateur iterateur = iterateur();
        int i = 0;

        // Parcourir tous les éléments jusqu'à arriver en fin de pile
        while(iterateur.possedeSuivant()) {
            etat[i++] = iterateur.suivant();
        }

        return etat;
    }
}
```

```java
package util;

/**
 * Classe itérateur qui pointe sur un élément.
 * @author Decoppet Joris, Ducommun Hugo, Martins Alexis
 */
class Iterateur {
    // Elément représenté par l'itérateur
    private Pile.Element element;

    /**
     * Constructeur de la classe Iterateur.
     * @param e Elément qui est représenté par l'itérateur.
     */
    public Iterateur(Pile.Element e) {
        this.element = e;
    }

    /**
     * Fait passer l'itérateur au prochain élément et
     * renvoie la valeur de l'élément actuel.
     * @return Valeur de l'élément courant de l'itérateur.
     * @throws NullPointerException Génère une exception s'il ne possède
     * pas d'élément à sa suite.
     */
    public Object suivant() {
        if(!this.possedeSuivant()) {
            throw new NullPointerException("Aucun élément suivant");
        }

        Pile.Element actuel = this.element;
        this.element = this.element.suivant;
        return actuel.donnee;
    }

    /**
     * Indique si l'itérateur possède un élément suivant.
     * @return Vrai --> l'itérateur possède un élément suivant.
     *         Faux --> l'itérateur ne possède pas d'élément suivant.
     */
    public boolean possedeSuivant() {
        return this.element != null;
    }
}
```

```java
package hanoi;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

/**
 * Classe de tests de la classe Hanoi.
 * @author Decoppet Joris, Ducommun Hugo, Martins Alexis
 */
public class HanoiTest
{
    /**
     * Vérifie que le statut des tours est correct.
     */
    @Test
    public void statusHanoiTest() {
        Hanoi h = new Hanoi(3);
        int[][] status = h.status();
        assertTrue(status[0][0] == 1 && status[0][1] == 2 && status[0][2] == 3 && status[0].length == 3 &&
                   status[1].length == 0 &&
                   status[2].length == 0);
    }

    /**
     * Vérifie que le programme se termine bien correctement.
     */
    @Test
    public void finishedHanoiTest() {
        Hanoi h = new Hanoi(3);
        h.solve();
        assertTrue(h.finished());
    }

    /**
     * Vérifie que le jeu n'est pas forcément résolu.
     */
    @Test
    public void notFinishedHanoiTest() {
        Hanoi h = new Hanoi(3);
        assertFalse(h.finished());
    }

    /**
     * On vérifie que le programme soit résolu correctement et que tous les éléments soient au bon endroit.
     */
    @Test
    public void solveHanoiTest() {
        Hanoi h = new Hanoi(3);
        h.solve();
        int[][] status = h.status();
        assertTrue(status[0].length == 0 &&
                   status[1].length == 0 &&
                   status[2][0] == 1 && status[2][1] == 2 && status[2][2] == 3 && status[2].length == 3);
    }
}
```

```java
package util;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

import java.util.NoSuchElementException;

/**
 * Classe de tests de la classe Pile.
 * @author Decoppet Joris, Ducommun Hugo, Martins Alexis
 */
public class PileTest
{
    /**
     * Vérifie que la pile vide est indiquée comme telle.
     */
    @Test
    public void pileVide(){
        Pile p = new Pile();
        assertTrue(p.estVide());
    }

    /**
     * Retirer un élément d'une pile vide provoque une erreur
     */
    @Test
    public void desempilerPileVide(){
        Pile p = new Pile();
        assertThrows(NoSuchElementException.class, ()->p.desempiler());
    }

    /**
     * Vérifie que la méthode retournant la taille donne bien la bonne taille.
     */
    @Test
    public void pileCompterElement(){
        Pile p = new Pile();
        p.empiler(3);
        p.empiler(3);
        p.empiler(3);
        assertEquals(3, p.obtenirTaille());
    }

    /**
     * Vérifie que les objets sont empilés, puis dépilés dans l'ordre.
     */
    @Test
    public void pileEnleverElement(){
        Pile p = new Pile();
        p.empiler(3);
        p.empiler(5);
        p.empiler(7);
        assertEquals(7, p.desempiler());
        assertEquals(5, p.desempiler());
        assertEquals(3, p.desempiler());
    }

    /**
     * Vérifie que l'affichage d'une pile se fait correctement
     */
    @Test
    public void pileAfficher(){
        Pile p = new Pile();
        p.empiler(3);
        p.empiler(5);
        p.empiler(7);
        assertEquals("[ <7> <5> <3> ]", p.toString());
    }

    /**
     * Vérifie que l'état de la pile est correct
     */
```

```
    @Test
    public void pileStatus(){
        Pile p = new Pile();
        p.empiler(3);
        p.empiler(5);
        p.empiler(7);
        assertEquals(7, p.etatPile()[0]);
        assertEquals(5, p.etatPile()[1]);
        assertEquals(3, p.etatPile()[2]);
    }

}
```

```java
package util;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

/**
 * Classe de tests de la classe Iterateur.
 * @author Decoppet Joris, Ducommun Hugo, Martins Alexis
 */
public class IterateurTest
{
    /**
     * Vérifie que la méthode suivant des itérateurs pase bien à l'élément suivant
     * en retournant l'élément courant.
     */
    @Test
    public void obtenirSuivantIterateur() {
        Pile p = new Pile();
        p.empiler(5);
        p.empiler(7);
        p.empiler(9);
        Iterateur it = p.iterateur();
        assertEquals(9, it.suivant());
        assertEquals(7, it.suivant());
        assertEquals(5, it.suivant());
    }

    /**
     * S'il n'y a plus de suivant, alors une exception devrait être levée.
     * La méthode possedeSuivant retourne false pour cela.
     */
    @Test
    public void obtenirSuivantVideIterateur() {
        Pile p = new Pile();
        Iterateur it = p.iterateur();
        assertThrows(NullPointerException.class, ()->it.suivant());
    }

    /**
     * Vérifie que la méthode possedeSuivant retourne true lorsqu'il y a un
     * élément suivant.
     */
    @Test
    public void aUnSuivantIterateur() {
        Pile p = new Pile();
        p.empiler(3);
        p.empiler(4);
        Iterateur it = p.iterateur();
        assertTrue(it.possedeSuivant());
    }
}
```