LOG3430
Méthodes de test de validation du logiciel

Travail pratique 1

Section N°1

Nicolas Richard – 1681198
Adrien Budet – 1721823
Chunxia Zhang – 1751567

Polytechnique Montréal
14 septembre 2015

Manuel d'utilisation

Deux méthodes existent pour interagir avec notre module Java : par la ligne de commande ou par l'appel de méthodes du paquetage depuis du code Java. Au cours des paragraphes qui suivent, nous explorerons ces deux manières.

Ligne de commande

Si l'on désire utiliser notre module depuis la ligne de commande, il suffit d'entrer la commande suivante au sein de l'interpréteur de commande

```
java SuiteChainee <Chemin> <Opérateur> <Val1> <Val2> <taille> <etatVide>
```

La signification des arguments est strictement identique à celle décrite au sein de l'énoncé du travail. Pour cette raison, nous éviterons de la réécrire ici. Cette commande aura le même effet que d'appeler le constructeur de la classe *SuiteChainee* : une suite de la taille spécifiée sera construite. À ce titre, cela constitue le principal désavantage de cette méthode d'interaction : une fois la commande appelée, il est impossible de modifier la suite avec un appel à *add(element)*. À noter que dans le cas des chaînes de caractères, il ne faut pas avoir aux guillemets pour les délimiter comme c'est le cas dans le code source Java. De plus, il faut avoir préalablement compilé les fichiers sources et il faut exécuter cette commande au sein du dossier contenant les fichiers *.class*.

Inclusion au sein de fichiers de code source Java

L'autre moyen d'utiliser notre module est d'importer ce dernier au sein d'un fichier de code source Java et de construire une instance de l'objet grâce à son constructeur. Le module contient une classe dénommée *SuiteChainee*. Il s'agit de la classe d'intérêt pour les utilisateurs du module. La signature du constructeur est identique à celle décrite dans l'énoncé du travail. Comme ce fut le cas précédemment, nous éviterons ici de la réécrire.

Manuel de conception

Lorsque vint le temps de concevoir notre module Java, nous avons fait plusieurs choix techniques en vue de résoudre les différentes contraintes imposées par l'énoncé initial du travail. Au cours de la prochaine section, nous détaillons et justifions ces derniers dans le but de donner une meilleure compréhension de notre travail à celui qui le consultera.

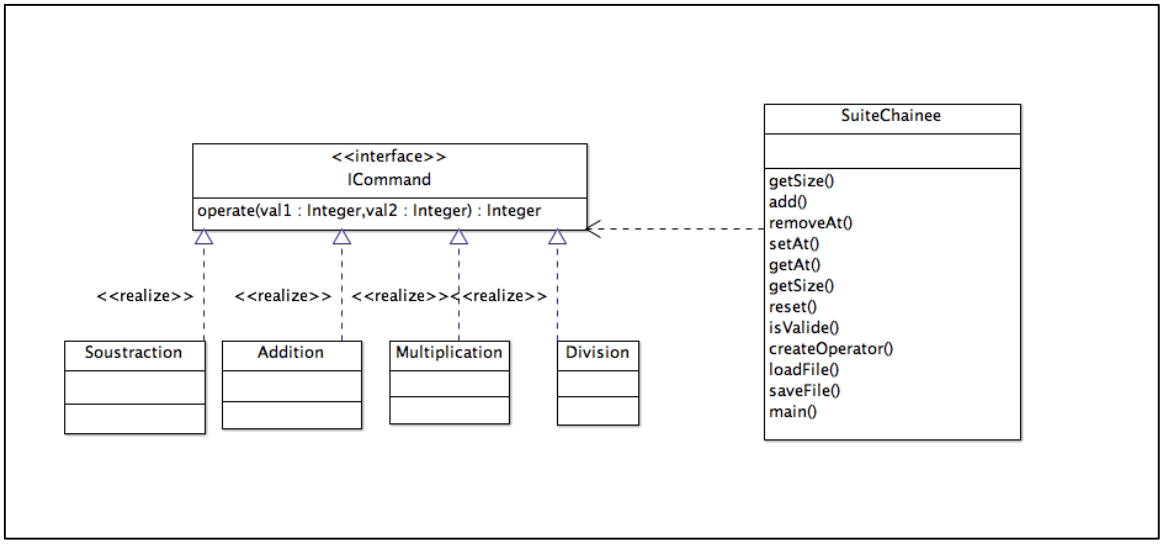

Figure 1 – Diagramme de classes approximatif du module

Ainsi donc, en premier lieu, nous avons décider de découpler de la classe principale de la suite les opérations mathématiques permettant d'en déduire les éléments suivants. Pour ce faire, nous avons créés une interface dénommée *ICommand* de manière à implémenter le patron *Commande* et pour chaque opération (addition, soustraction, multiplication, division), nous avons créé une classe correspondante implémentant cette dite interface. À ce titre, pour faciliter la création de suites de tests visant la construction de la suite, il est possible d'assigner, pour chaque opérateur, la classe liée à ce dernier. Du coup, de par cette architecture, il devient aisé d'ajouter de nouvelles opérations dans le futur en modifiant une quantité très limitée de code. De même, réaliser des tests sur cette partie du module devient bien plus aisé. En effet, si ces opérations mathématiques auraient été implémentées sous la forme de méthodes privées au sein de la classe *SuiteChainee*, il

aurait été impossible de les tester à l'aide de méthode standards au sein d'un cadre d'application tel que *JUnit*. Évidemment, cette limitation aurait pu être contournée en utilisant des méthodes protégées ou publiques, mais du coup, l'interface de la classe se serait vu polluée par des méthodes utilitaires ayant peu d'intérêt aux yeux des utilisateurs.

Dans un autre ordre d'idée, en ce qui concerne l'implémentation des nœuds de la liste chainée, nous avons pris la décision de créer une classe *Node* qui encapsule les fonctionnalités requises d'un tel nœud et de l'imbriquer directement au sein de la classe *SuiteChainee*. En effet, en procédant de la sorte, nous regroupons tout le code lié au sein d'un même fichier, ce qui permet d'accroître la cohésion du module ; les nœuds n'étant pas destinés à être utilisés par une autre entité que la classe *SuiteChainee*, il est inutile de leur donner une classe en bonne et due forme.

En ce qui a trait à la lecture et à l'écriture du fichier, nous avons décidé d'utiliser une solution existante qui est fournie au sein même du *Java Development Kit*, la classe *Properties.* En effet, en plus d'être plus fastidieuse à mettre en place, une solution de notre cru aurait nécessité d'élaborer une série de tests en vue de nous assurer de son bon fonctionnement. En utilisant une classe utilisée par un nombre incalculable de personnes, nous pouvons assumer qu'elle fut testée rigoureusement au cours des années, ce qui nous évite une grande quantité de travail.

Par ailleurs, sur un plan plus technique, nous stockons la taille de la liste au sein d'une variable privée. Ainsi, lorsqu'une personne fait appel à la méthode *getSize()*, la complexité asymptotique de l'opération est constante au lieu d'être linéaire. Aussi, en ce qui concerne la méthode *removeItem*, nous avons pris la décision de ne supprimer que la première occurrence de l'élément dans le cas où on retrouverait le même nombre plusieurs fois.

Annexe

Pour ordonnancer les propriétés au sein du fichier **.properties*, nous avons eu recours au code suivant, rédigé par l'utilisateur *Steve McLeod*, qui fut trouvé à cette adresse URL : http://stackoverflow.com/questions/17011108/how-can-i-write-java-properties-in-a-defined-order

```
Properties tmp = new Properties() {
    @Override
    public synchronized Enumeration<Object> keys() {
        return Collections.enumeration(new TreeSet<Object>(super.keySet()));
    }
};
```