Cours INF-7845
Travail Pratique : Ordre partiel avec Nit

Alexandre PENNETIER

Mars 2012

1 Rappel du sujet

Le travail demandé consiste en la réalisation d'un ordre partiel, sous forme d'une API (Application Programming Interface) réalisée en langage Nit.

Un ordre partiel est un ensemble d'éléments reliés entre eux par des relations binaires réflexives, transitives et antisymétriques.

La bibliothèque doit modéliser ce comportement et proposer certaines fonctionnalités pour leur manipulation.

2 Conception

2.1 Classes de la bibliothèque

La première est la classe de l'ordre partiel : PartiallyOrderedSet.

Elle stocke la liste des éléments et assure toute la logique de manipulation des éléments et des relations. L'utilisateur de la bibliothèque est en contact permanent avec cette classe, à chaque fois qu'il souhaitera manipuler des éléments.

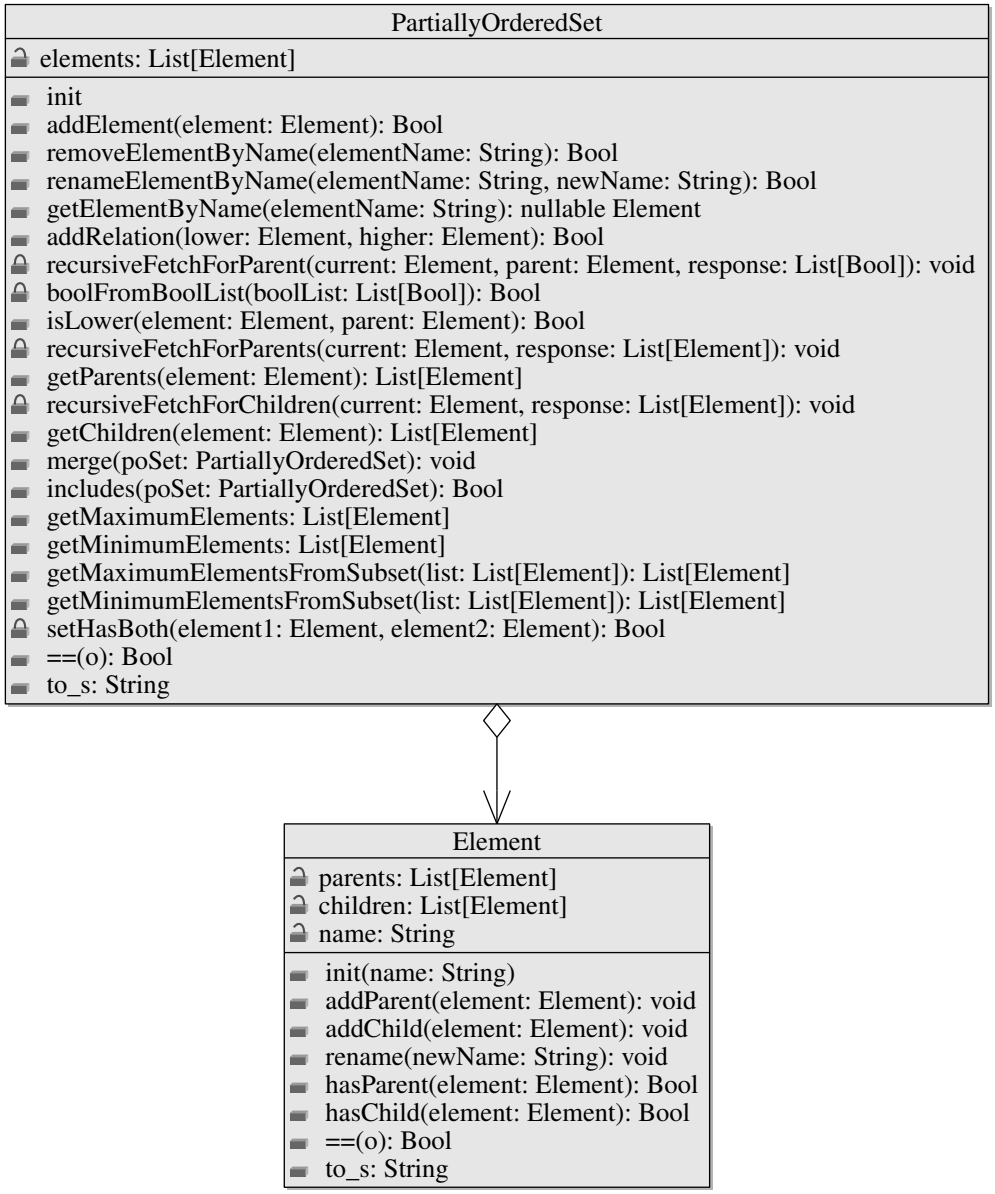
Ainsi, lorsque l'utilisateur souhaitera ajouter une relation entre deux éléments, c'est à l'objet de l'ordre qu'il s'adresse. Celui-ci s'occupe alors de la cohérence des données contenues dans ses éléments.

La seconde est un élément : Element.

Le choix a été fait d'inclure les relations dans cette classe. Ceci implique certains avantages et inconvénients qui sont détaillés dans la section suivante.

Les comparaisons se font sur le nom des éléments et le nom de ses parents et enfants. Ainsi, il est nécessaire de prendre des mesures contre l'ajout d'éléments redondants. Un élément ne peut être ajouté s'il existe déjà dans l'ordre.

2.2 Diagramme de classes



2.3 Avantages et inconvénients de la solution

Le grand choix à déterminer lors de la conception était l'emplacement des relations directes entre les éléments : doit-on les stocker dans l'ordre ou dans ses éléments ?

Le choix s'est porté vers le stockage des relations dans les éléments, à la fois pour les enfants et parents directs (à la manière d'une liste doublement chaînée). Le parcours récursif de l'ordre entraine le parcours de plusieurs petites listes plutôt qu'une grande. Ainsi, chaque opération se fait à un niveau plus atomique. Chaque élément est garant de ses relations avec les autres.

Selon les fonctionnalités demandés, la balance penche en faveur de ce choix ou de l'autre. Par exemple, lorsqu'on doit déterminer si un élément est maximal ou minimal dans l'ordre, si les relations étaient stockées dans l'ordre, il faudrait parcourir toutes les relations pour chaque élément. Si elles sont stockées dans l'élément, il faut simplement regarder si la liste de ses enfants ou de ses parents est vide. C'est la même logique pour vérifier si deux éléments sont parents directs : un coup d'oeil sur la liste d'un des deux éléments.

Tous ces parcours de listes sont avantageux car, dans la plupart des situations, elles restent de taille raisonnable, contrairement à une liste globale des relations, qui peut prendre de grands proportions, si l'ordre devient volumineux.

Par conséquent, le parcours récursif de l'arbre à la recherche de relations transitives est bien plus efficace que le parcours de la liste des relations. Avec une liste contenant toutes les relations, le saut d'un élément à l'autre implique de parcourir, dans le pire des cas, toutes les relations de l'ordre. La pire complexité qu'on puisse avoir est donc entre deux éléments reliés qui sont aux deux extrémités de l'arbre. Dans ce cas elle est quadratique. Avec la solution choisie, on ne réduit pas le nombre de sauts mais on réduit la longueur des listes à parcourir.

D'un autre côté, lors d'une fusion entre deux ordres, une simple comparaison entre les relations de l'ordre serait plus simple. Ici, il faut scruter chaque élément individuellement et vérifier à chaque fois ses parents et enfants avant toute écriture afin de ne pas créer d'inconsistance et de respecter les règles lors des conflits.

L'ajout d'une relation est également un peu plus compliqué. Il faut partir à la recherche des éléments mis en relation, pour pouvoir leur attribuer individuellement leur nouveau parent/enfant.

Également, lors de la comparaison entre deux éléments, pour être

sûr qu'ils sont égaux, il est nécessaire de regarder leurs parents et enfants, là où le système stockant les relations au niveau de l'ordre, la comparaison n'aurait lieu que sur l'élément seul.

Ainsi, de façon ponctuelle, le fait de stocker les relations au sein des éléments complexifie légèrement le travail du développeur mais, sur la plupart des fonctionnalités, il le simplifie et rend la bibliothèque plus efficace temporellement et sans coût spatial.

On aurait pu aussi se contenter de stocker les relations dans un seul sens : stocker uniquement les parents ou les enfants. Ceci permettrait de diviser par deux la quantité de données nécessaires pour le stockage des relations. Néanmoins, cette information manquante représente un coût en termes d'efficacité temporelle.

Exemple : retirer les informations sur les enfants réduit grandement la simplicité des opérations telles que la recherche des éléments minimaux ou la recherche de tous les éléments inférieurs à un élément.

3 Le langage Nit

La syntaxe, maintenant maitrisée, s'avère toujours aussi agréable. Je me prends à pester contre Java devant tant de verbosité et à regarder Ruby avec envie.
Petite note sur le typage adaptif du type « nullable » qui est pratique à utiliser.

4 Utilisation de notions abordées en cours

L'héritage n'a toujours pas été utilisé entre les classes du projet. Seulement deux classes sont présentes et la généricité a été un allier suffisant.

La généricité a donc été largement employée dans le stockage des données (donc en tant qu'utilisateur). Les deux classes stockent des éléments sous formes de listes. Les situations vues en cours ont apporté un éclairage sur le concept mais, sans la mettre en danger avec de l'héritage, je n'ai pas eu l'occasion de froisser le compilateur.

Note : les sources sont accessibles sur ce dépôt :
`git://github.com/Dresdof/POSetNit.git`