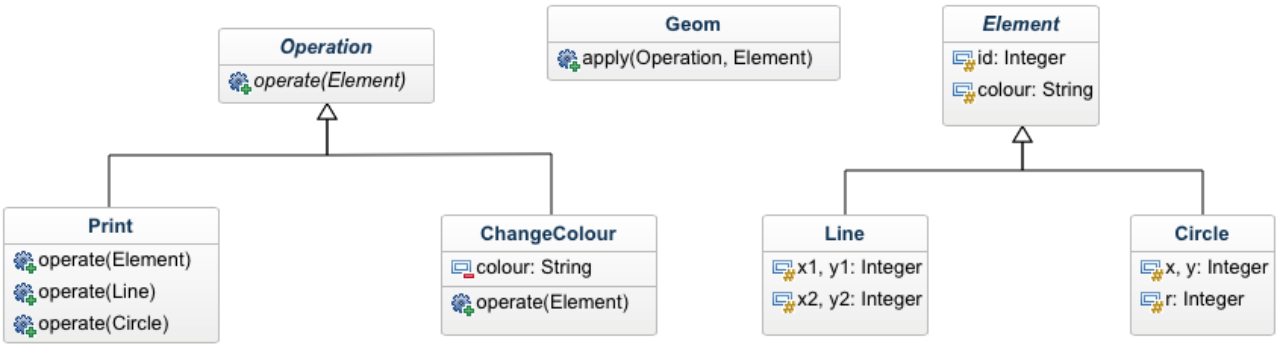
Dessins vectoriels

POGL, TP7. Liaison dynamique, visiteurs, structures composites

Dans ce TP, nous allons utiliser des techniques objet pour représenter et manipuler des dessins vectoriels.

Surcharge statique et liaison dynamique

Le squelette de code fourni (`http://www.lri.fr/~blsk/POGL/TP7.zip`) réalise le diagramme de classe suivant, dans lequel on décrit plusieurs types d'éléments graphiques et d'opérations pouvant s'y appliquer.



La classe `Geom` fournit une méthode `apply` qui applique une opération à un élément graphique.

1. Dans la classe `Geom`, écrivez une méthode `main` qui crée un objet `o` de classe `Print`, un objet `e` de classe `Line`, et réalise les deux appels `o.operate(e)` et `apply(o, e)` (notez que ce dernier consiste aussi essentiellement en un appel à `operate`).
   Dans chacun des cas, quels sont les types apparents de `o` et `e` au niveau de l'appel à `operate` ? quels sont les types réels de `o` et `e` ? Quelle méthode `operate` est sélectionnée ? Qu'en déduire sur la manière dont Java utilise les types apparents et types réels pour déterminer la méthode à appeler ? Testez quelques autres variantes avec des conversions de type pour confirmer.
2. Comme nous venons d'établir que la surcharge statique servait surtout à ajouter de la confusion, modifiez la classe `Print` en renommant `operateLine` la méthode `operate` attendant un paramètre de type `Line` et `operateCircle` la méthode `operate` attendant un paramètre de type `Circle`. Dorénavant on évitera systématiquement la surcharge en utilisant cette technique.

Sélection en fonction du type réel de tous les paramètres

Nous allons maintenant chercher à obtenir des méthodes `operate*` qui sont sélectionnées en fonction des types réels à la fois du paramètre implicite et des paramètres explicites. Ceci permettra notamment d'ajouter des opérations dont le comportement s'adapte à chaque type d'élément.

Solution 1 : réflexion et transtypage explicite. La solution la plus brutale consiste à introduire des tests utilisant `instanceof` et du transtypage.

3. Appliquez cette solution pour créer une nouvelle classe d'opération `Translate`, qui applique une translation de coordonnées `dx, dy` à l'élément auquel elle est appliquée.

Solution 2 : liaison dynamique double. Pour éviter de faire soi-même le test et la sélection d'un comportement, on peut demander à l'opération de faire appel à des méthodes déclarées dans `Element` et (re)définies dans chaque classe fille.

4. Appliquez cette solution pour créer une nouvelle classe d'opération `Zoom`, qui multiplie les coordonnées et les distances par un facteur `z`, en faisant appel à une méthode `zoom` déclarée (abstraite) dans `Element` et définie dans `Line` et `Circle`.
5. Modifiez de même l'opération `Print` pour qu'elle affiche la nature réelle, l'identifiant et la couleur de l'élément ciblé.

Remarquez que cette technique demande, pour chaque opération, d'écrire du code dans chaque classe d'`Element`.

Solution 3 : visiteurs. Le *design pattern* Visiteur donne une solution générique qui ne demande de modifier les classes d'`Element` qu'une seule fois. Le principe est le suivant :

— La classe `Operation` implémente l'interface `Visitor` suivante, dans laquelle un traitement est prévu pour chaque type d'élément.

```
interface Visitor {
    public void visitLine(Line l);
    public void visitCircle(Circle c);
}
```

Les méthodes `visitLine` et `visitCircle` sont à définir dans chaque classe concrète d'`Operation`.

— La classe `Element` implémente l'interface `Visitable` suivante.

```
interface Visitable {
    public void accept(Visitor v);
}
```

Pour cela, chaque classe concrète d'`Element` fournit la définition

```
public void accept(Visitor v) {
    v.visit***(this);
}
```

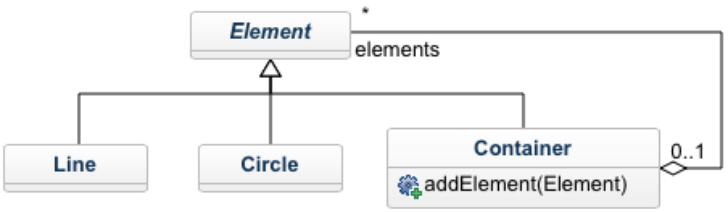
en remplaçant `visit***` par la désignation de la méthode `visit` adaptée à la classe concrète [1].

Ainsi, c'est au niveau des classes implémentant `Visitor` que l'on place tout le code à appliquer aux différents `Element` : les définitions sont groupées par opération plutôt que par objet cible.

6. Modifiez les classes d'`Operation` et d'`Element` pour qu'elles implémentent respectivement les interfaces `Visitor` et `Visitable`.

Structure composite

Nous allons maintenant tenir compte du fait qu'un dessin vectoriel peut contenir plusieurs éléments, et que les éléments peuvent être groupés. Pour ceci on ajoute une nouvelle sous-classe d'`Element` appelée `Container`, qui contient une liste d'éléments. Notez qu'il est tout à fait possible qu'un `Container` contienne d'autres `Container`, et ce jusqu'à une profondeur arbitraire. Nous utilisons là le *design pattern* Composite.



Les visiteurs donnent une manière agréable d'itérer une opération sur tous les éléments d'une collection, même si cette collection a une structure potentiellement complexe comme peut l'être un enchevêtrement d'`Element` et de `Container`. Le principe est de séparer deux aspects :

— L'opération à effectuer, qui est définie dans le `Visitor` comme on l'a déjà vu.

— Le parcours de la collection d'objets, qui est défini une fois pour toutes dans les `Visitable`.

Ainsi, c'est la méthode `accept` de la classe `Container` qui va se charger de propager l'opération aux `Element` qu'elle contient, en faisant appel à leurs propres méthodes `accept`.

7. Ajoutez une classe `Container` telle que décrite ci-dessus, et définissez sa méthode `accept` de sorte à ce qu'un visiteur atteignant un `Container` soit propagé aux `Element` qu'il contient.
8. Ajustez l'interface `Visitor` et les classes implémentant `Visitor` pour tenir compte de ce troisième type d'`Element`.
9. Créez un nouveau visiteur, qui parcourt une hiérarchie d'`Element` et affiche le code SVG correspondant.
   `https://developer.mozilla.org/fr/docs/Web/SVG/Tutoriel`

1. Ce *design pattern* est généralement présenté en donnant le même nom `visit` à toutes les méthodes `visit***`. On remarquerait alors ici que la méthode `accept` est recopiée à l'identique dans chaque classe concrète (mêmes nom, signature et code) plutôt que factorisée dans la classe abstraite mère. Si vous pensez que c'est bizarre, revenez à la question 1.