Programmation avancée

Décorateurs python

Ils permettent de décrire le fonctionnement d'un objet python.

```
def attention(f):
    @functools.wraps(f)
    def f_interne(*p,**k):
    print('Debut de f_interne()')
    f(*p,**k)
    print('fin de f_interne()')
    print('dans attention')
    print(func.__name__ + " a été appelée")
    return f_interne

@attention
def f(x, y=4):
    print(x-y)
```

Inversion de contrôle

L'IoC est un patron d'architecture, une entité prend le contrôle du programme. Un module peut très bien appeler le programme principal pour lui demander des infos.

Les Annotations java

Les annotations java sont comme les décorateurs python. Ils sont lu à la compilation, les méta-données sont enregistrées dans les fichiers de classes et précisent les intéractions avec les méthodes / attributs. Exemple : @Override (marqueur) et Annotation (multi) paramétrées.

Proxy

Un proxy (ou mandataire ou procuration) est un intermédiaire. C'est un objet que l'on va renvoyer à la place de l'objet réel, qui permet d'autres actions comme des logs, des vérification ou même de la sécurité (exemple du proxy utilisé lors du tp du jeu des allumettes).

Il existe des proxy build-in, par exemple le concept de unmodiableList / unmodifiableCollection dans java, qui permet de retourner un jeu de donnée inaltérable. Ou encore des vérification sur les données comme chekedList / checkedMap.

PLus généralement, on définit un proxy en écrivant l'interface correspondante et en interceptant les appels, on peut ensuite vérifer que l'action est autorisée, on peut renvoyer une copie de la méthode réelle, on peut faire des vérification de type, etc...

Proxy et introspection

L'introspection permet à un programme java de découvrir dynamiquement des informations sur une classe.

La `class` java `Proxy` permet de décrire des proxy avec ce principe d'introspection. On définit un objet réalisant l'interface `InvocationHandler` qui va traiter tous les appels de méthode. Cet objet renvoi une methode dite `invoke` qui prend en paramètres la méthode appelée et ses arguments, un dernier parametre permet de déterminer quel proxy est à l'origine de cet appel. Dans la pratique, on utilise la classe `Proxy`

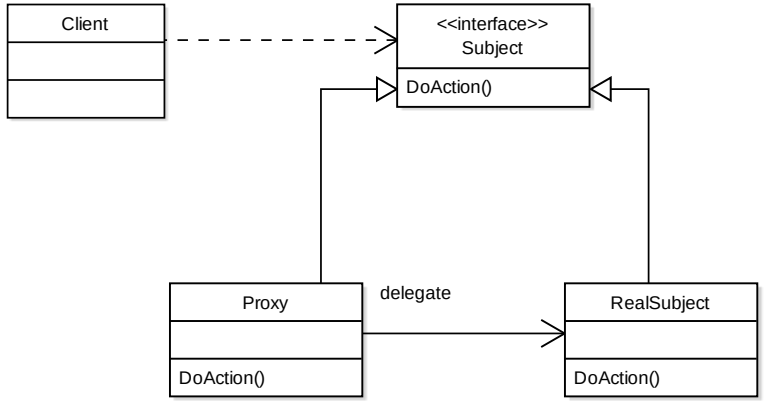

Figure 1: Alt text here

avec `newProxyInstance`qui prend en argument : le loader de la classe concerné (List.class.getClassLoader() par exemple), les interfaces concernées (objets de type class) et le InvocationHandler qui s'en occupera).

Intérêts : créer des proxy dynamiquement à l'execution, code + léger, même invocationHandler pour plusieurs proxy. Inconvénient : le recours à l'introspection est non négligeable en terme de temps de d'éxecution.

En Python : on a les méthodes : getattr(object, name), hasattr(object, name), setattr(object, name, value), delattr(object, name).

Programmation par aspect

Elle permet de régler des préocupations transversales comme la sécurité, la persistance ou le logging.

Enchevêtrement du code : Faire du non metier autour du metier : il y en a partout.

Eparpillement de code : Être obligé de dupliquer les même préoccupations dans différents contextes (classes).

On créé toutes les preoccupations, ce sont les règles de la programmation par aspect qui va régir leur intéractions.

aspect : Un aspect est une entité logicielle qui capture une fonctionnalité transversale à une application.

point de jonction : Un point de jonction est un point dans le flot de contrôle d'un programme dans lequel un ou plusieurs aspects peuvent être appliqués. (méthode, attribut, exception, constructeur)

coupe : Une coupe sélectionne un ensemble de points de jonction.

code advice : Un code advice est un bloc de code définissant le comportement d'un aspect.

mécanisme d'introduction : Le mécanisme d'introduction est un mécanisme d'extension permettant d'introduire de nouveaux éléments structuraux au code d'une application.

tissage : Le tissage (weaving) est le processus qui prend en entrée un ensemble d'aspects et une application de base et fournit en sortie une application dont le comportement et la structure sont étendus par les

aspects.

```
// Voici une coupe nommée
pointcut move():
call(void FigureElement.incrXY(int,int)) ||
call(void Point.setX(int))               ||
call(void Point.setY(int))               ||
call(void Line.setP1(Point))             ||
call(void Line.setP2(Point));

// On peut se servir d'expression régulières
call(void Point.set*(..))
call(public * Line.* (..))

// Code advice : before, around ou after

before() : move() { System.out.println("Figure sur le point d'être déplacée"
around() : call(Display.update()) { if (! Display.disabled()) proceed();}
// Le proceed redonne la main au programme

// Ajouter des éléments de classe
public String Point.name ;
public void Point.setName ( String name ) { this.name = name ; }
public String Point.getName ( ) { return name ; }

// Changer les relations entre classes
declare parents : (Point || Line) extends GeometricObject ;

// Un aspect d'update d'affichage :
public aspect UpdateDisplay {
    pointcut move(FigureElement elem) : target (elem) &&
( call ( void Line.setP1 (Point) ) ||
  call ( void Line.setP2 (Point) ) ||
  call ( void Point.setX (int) )   ||
  call ( void Point.setY (int) )   ||
  call ( void FigureElement.incrXY(int, int) ) ) ;

    after(FigureElement elem) returning : move ( elem ) {
        Display.update(elem);
    }
}
```