Université catholique de louvain-la-neuve

LSINF2335 - Programming Paradigms

Context-Oriented Ruby project

Aimable Hirwa
Jos Zigabe

Groupe H

13 mai 2016

1 Introduction

Dans le cadre du projet *Context-Oriented Ruby* du cours Programming Paradigms, il nous a été demandé de créer un COP (Context-Oriented Programming) en Ruby à l'aide des outils de réflexivité que ce langage offre.

Dans ce rapport, on va décrire comment on a implémenté notre COP et les différents choix de conception que nous avons effectués.

Pour réaliser notre COP nous nous somme fortement basé sur les solutions de séances de TPs. En effet, elle offré une solution fiable et correcte.

2 Fonctionalities

2.1 Example

```ruby
#import file
require "../COP/Context/context.rb"
require "../COP/Phone/phone.rb"
require "../COP/Phone/phoneCall.rb"

# name the contexts
@quietContext = Context.named("quiet") # Context named quiet
@screeningContext = Context.named("screening") # Context named screening
# adapt the class phone to contexts
@quietContext.adaptClass(Phone, "advertise", Phone.advertiseQuietly)
@screeningContext.adaptClass(Phone, "advertise", Phone.advertiseWithScreening)

phone = Phone.new # new phone object
call = PhoneCall.new # new phoneCall object
call.from = "Alice" # assign call to Alice
phone.receive(call) # phone receive a call

puts phone.class.advertise # out : "ringtone" the default behaviour
@quietContext.activate # quietContext is actived
puts phone.class.advertise # out : "vibrator" the quiet behaviour
@quietContext.deactivate # quietContext is deactived
puts phone.class.advertise # out : "ringtone" back to default behaviour
@screeningContext.activate # screeningContext is actived
puts phone.class.advertise # out : "ringtone with screening" the screening behaviour
```

3 Architecture

L'architecture du COP est composé de 5 classe et tout un ensemble de modules qui viennent completer ces classes.

3.1 Context

La classe `Context` permet d'avoir le context comme une entité *first-class*. Elle contient un ensemble de modules et chacun de ces modules à des méthodes spécifiques lié à celui-ci. Les raisons qui nous ont pousseé

à faire cella était que nous ne voulions pas avoir une classe `Context` contenant un trop méthodes et rendant la classe moins lisible. Alors grace au *mixin modules* de Ruby nous avons pu repartir ces différentes méthodes dans des modules pour ensuite les inclures dans la classe `Context` afin d'avoir une meilleuré lisibilité. Une autre alternative aurait été l'héritage multiple mais Ruby ne le permet pas donc la meilleure solution pour nous été d'utiliser le `mixin modules`.

Initialization & singleton class

On y retrouve tout d'abord dans la classe `Context` une méthode initialisation : `initialise()` qui va initiliser les differents attributs de la classe `Context` et on retrouve également une classe anonyme qui représente le singleton. Cette dernière s'occupe de definir le context par defaut.

On aura ensuite un ensemble de modules qui ne sont que des parties de codes, qui vont venir enrichir la classe `Context` :

Accessing module

Dans ce module on a 3 méthodes : `manage()`, `name()` et `name(aString)` qui seront des getters et setter pour `Context`. La méthode `manage()` n'est pas un simple getter étant donné qu'elle permet d'assigner un manager si le context n'en posséde pas encore. Si le context par défaut à définit un manager alors le mansager du context courrant devient le manager du context par défaut sinon un nouveau manager.

Activation module

Ce module contient les méthodes qui vont être utilisé par `Context` afin d'activer et de désactiver un context en exécution. On retrouve ainsi les méthodes : `activate()`, `desactivation()`, `isActive()` et `ActivationAge`

Adaptation module

Dans ce module on retrouve une méthode importante pour le COP : `adaptClass(aClass, aSelector, aMethod`. Cette méthode permet de modifier le comportement d'une méthode appeler `aSelector` de la classe `aClass`. C'est l'adaptation de la méthode `aSelector` au comportement d'une autre méthode `aMethod`.

Life cycle module

Ce module est celui de la méthode `discard()` qui se charge d'éjécter le context `self` qui fait appel à lui.

Printing module

Ici on retrouve simplement la méthode `printOn()` qui se charge d'afficher le contenu de la classe.

Private

Enfin on a un ensemble de méthode privée tels que `activationCount()`, `activateAdaptation()` et `roolbackAdaptation()` qui pour les deux premiers sont des getter et pour le dernier une méthode qui permet d'annuler l'activation d'un context si un de ces adaptations n'est pas activate.

3.2 ContextManager

Cette classe s'occupe de la gestion des contexts et des adaptations et elle aussi est composé de plusieurs modules.

Accessing module

Dans ce module on y retrouve 4 méthodes : `activeAdaptations()`, `activeContext()`, `directory`, `resolutionPolicy` et `resolutionPolicy` qui seront des getters et setters de `ContextManager`.

Activation module

Ce module contient les méthodes permettant au `ContextManager` d'activer et de désactiver un context donné. On retrouve ainsi les méthodes : `contextActivationAge(aContext)`, `signalActivationRequest(aContext)` et `signalDeactivationRequest(aContext)`

Adaptation module

Dans le module adaptation on retrouve les méthodes `activateAdaptation(aContextAdaptation)`, `adaptationChainFor(aSelector)`, `deactivateAdaptation(aContextAdaptation)` et `deployBestAdaptationForClass(aClass, aSelector)`. Ces méthodes permettent la gestion des comportements pour les context. Elles permettent d'activer, de désactiver et d'adapter une méthode donné d'une classe donné.

Life cycle module

Tout comme le module du même nom utilisé par la classe Context, Life cycle contient une méthode qui se charge de supprimer un context non actif avec la méthode `discardContext()`.

Printing module

Ici on retrouve de nouveau simplement une méthode `printOn()` qui se charge d'afficher le contenu de la classe.

Resolution module

Ce regroupe un ensemble tels que `activationCount()`, `ageResolutionPolicy()`, `defaultRésolutionPolicy()`, `findNextMethodForClass(aClass, aSelector, aMethod)`, `noRésolutionPolicy()` et `singleAdaptationRésolutionPoli`
Les méthodes de ce module s'occupe de résoudre des problèmes d'ambiguités liés aux adaptations lorsque plusieurs choix sont possibles.

Private

Pour finir dans cette classe `ContextManager` on retrouve deux methode privée : `activeAdaptation(aCollection)` et `directory(aDirectory)`

3.3 ContextAdaptation

Tout comme les deux autres classe, `ContextAdaptation` est également composée de modules.

Accessing module

Le module Accessing regroupant des accesseurs et mutateurs pour la classe `contextAdaptation`

Installation module

Ce module contient la méthode `deploy)` qui se charge de redéfinir le comportement d'une méthode par le comportement d'une autre.

Printing module

Dans celui-ci on retrouve de nouveau simplement une méthode `printOn()` qui se charge d'afficher le contenu de la classe.

Testing module

Enfin, le module Testing regroupe des méthodes booleanne qui permettent d'effectuer desc comparaisons sur les adaptations : `adaptClass(aClass, aSymbol` et `sameTarget(aContextAdaptation`.

3.4 Phone

Il s'agit ici d'une classe qui représente un téléphone :

Accessing module

Ce module contient les accesseurs et mutateurs de la classe `Phone`.

Call handling & Call handling shortcuts module

Quant à ces modules, il regroupe toutes les méthodes qui servent à la gestion d'un appelle que ce soit pour la réception de l'appelle ou pour terminaison d'un appel.

3.4.1 PhoneCall

`PhoneCall` est la classe qui représente les appels téléphoniques. Elle contient une méthode qui permet de définir le nom de l'émetteur et un méthode pour y accéder. Une méthode `printOn()` permet d'affichier le contenu de la classe.

4 Design

Lors de la réalisation du COP, il a fallu considerer plusieurs comportement que l'on va décrire dans cette section.

4.1 COP Infrastructure

On a tout d'abord commencé par définir le context comme une entité *first-class* en créant la classe `Context`. Cet entité représente le context dans lequel le téléphone peut être. En changeant le context le téléphone doit être adapter au changement du context. Cette classe va donc permettre de définir différents contexts et pour chaque context de définir des adaptations liées à différents téléphones.

4.1.1 Default context

Losrqu'un context par défaut n'a été définit alors il faut en définir un en fessant appele à la méthode `default()` de la classe `Context` ou alors désigner un context existant comme étant le context par défaut. En quelque sorte, il doit toujours y avoir un context par defaut représentant les comportements par défauts des téléphones lorqu'elles n'ont pas étés adapter. Pour la gestion du context par defaut, nous avons utilisé une classe `singleton` qui est une classe anonyme en ruby non accessible par un object d'instance mais plutôt par `Context.default`. Cette classe *singleton* permet de pouvoir faire appel au context par défaut et d'en définir un si il n'y en a pas.

```ruby
class Context
   #Singleton class
   class << self
     # getter
     def default()
       if @default == nil
         @default = self.new
         @default.activate
       end
       return @default
     end
     #setter
     def default=(aContext)
       @default = aContext
     end
     #set the name of the context
     def named(aString)
       _ctx = self.new
       _ctx.name= aString
       return _ctx
     end
   end
end
```

4.1.2 Context life cycle management

Dans un second temps, il a fallu prendre en compte qu'un `Context` peut être écarté et qu'il ne pourra plus être activé. Nous avons donc dû creer une méthode `discard` dans la classe `context`. Cette méthode s'occupe dans un premier temps de demander au `ContextManager` que l'on va définir plus bas, d'écarter le context courant et ensuite il doit vérifier qu'il ne s'agit pas du context par défaut à fin de le définir à `nil` pour pouvoir retirer toutes les adaptations faites pour ce context.

4.1.3 Multiple activation

Ensuite il a fallut considerer le fait qu'un `Context` peut être activé/désactivé plusieurs fois. Pour résoudre ce sous problème nous avons défini dans la classe `Context` les méthodes `activate` et `deactivate`. La première se charger d'activer le context en envoyant tout d'abord un message d'activation au `ContextMAnager` qui lui va activer l'adapation la plus apprié du context et va ensuite deploye cette adaptation. La seconde méthode `deactivate` va désactiver l'adaptation du context en enlevant l'adaptation de la liste des adaptations actives et va ensuite déployer la dernière adaptation pour la méthode adapté.

4.2 Behaviour Adaptation

Afin que notre système soit *Context-Oriented*, il nous à fallu implémenter des méthodes qui se chargent d'adapter le système à différents contextes et nous expliquons nos choix dans les sous sections qui suivent :

4.2.1 Telephony framework

Pour cette partie du projet on a dû tout d'abord reproduire le `telephony framework` que l'on nous a fourni lors des sessions pratiques en Ruby. Dans notre solution on a gardé que deux classes : `Phone` et `PhoneCall`.

Dans la classe `Phone` on a utilisé une classe singleton pour y définir les méthodes *advertise* nécessaires pour avertir l'utilisateur d'un appelle. Ce sont les méthodes qui peuvent être adapter pour exécuter le comportement des autres.

4.2.2 Adaptation class

La méthode `AdaptClass` se trouve dans la classe `Context` et prend en 3 parametres qui sont sont la classe qui dans notre cas est `Phone`, un selector qui est un string représentant la méthode à adapter et une méthode qui est le nouveau comportement du selector. La méthode `AdaptClass` permet de créer une adaptation à partir des parametres et de l'enregistrer. Il enregistre une adaptation qui garde comportement par défaut et une adaptation qui est le nouveau comportement. Cella permet de pouvoir revenir au comportement par defaut de la méthode adapté. L'adaptation est directement activé si le context est activé. Attention qu'une adaptation pour un context ne pourra être accépté que si le context ne contient pas déjà une adaptation pour la classe et la méthode adapté.

4.3 Behaviour composition

A cet étape du projet il a fallu inclure des mécanismes de composition à notre système. En effet, le système devrait permettre à une méthode adapté d'invoquer la méthode précédent afin que le comportement adapté soit défini comme une extention du comportement précédent. Il faut demandons à la classe qui contient le nouveau comportement de faire appele au comportement précédent de la méthode à adapter. Pour cella nous avons une méthode `proceed` dans la classe phone qui permet de revoyer le comportement précédent. Le problème de cette solution c'est qu'elle ne permet pas de pouvoir récuper exactement le comportement précédent. Dans la solution la méthode `proceed` renvoi une valeur hardcodé. Cella à permis de passer les tests tels que ceux décrit dans pharo. Nous avons tenter d'implémenter une meilleur solution mais cette solution n'a pas été un succées. L'idée de cette solution est que la méthode qui contient le nouveau comportement fasse appele à une méthode `proceed` qui se trouverai dans la classe `Context`. A partir de cette classe il faut assayer de retrouver le précédent comportement. Pour cella, nous avons définit une variable de classe qui contiendrai tous les adaptations et à partir de cette liste d'adaptations retrouver le précédent adaptation du système. Ici dans la parti composition, le précédent comportement est le comportement par défaut étant donné qu'on peut activer qu'une adaptation à la fois pour une méthode à adapter.

4.4 Behaviour resolution

Sachant que la résolution demande la réalisation de la composition nous n'avont pas passé tous les tests pour cette partie. Mais nous avons quand même implémenter le comportement qui doit satisfaire la résolution. On sait que pour la résolution on veut pouvoir activer plusieurs adaptations à une même méthode. Pour ce faire, il faut avoir la possibilité de récupérer le dernier comportement. Nous avons donc mis les adaptations dans une liste d'adaptation ordonné suivant le moment ou l'adaptation à été activé pour facilité la reconnaissance du précédent adaptation. Une fois retrouver il faut le déployer pour l'adapter à la méthode à adapter.

5 Metaprogramming in Ruby

5.1 Self

`self` est le *receiver* par défaut pour les méthodes d'apppels. Ainsi si j'appelle une méthode sans donner un *receiver* explicit Ruby va chercher la méthode dans l'objet référencé par `self` pour cette méthode. Dans notre solution à différents endroits nous avons dû faire appel à ce keyword `self` ou `@var` afin de pouvoir traiter correctement les differents context.

5.2 Anonymous class

Une classe anonyme est aussi connu comme *singleton class* ou encore *metaclass*. Dans notre solution nous avons dû utiliser cette caractéristique de ruby afin de pouvoir notammant modifier le comportement de la classe `Phone` pour qu'il puisse s'adapter au different contexts.

5.3 Introspection

En Ruby, il est possible de récolter des informations sur une classe ou un objet durant son execution. Pour cela Ruby dispose des méthodes tels que `respond_to?` qui est un exemple d'introspection ou de reflexion que nous avons souvent utilisé dans notre solution :

```
# les appels rater sont retire des appels entrant pour etre mis dans les appels rate
def miss(aPhoneCall)
  if @incomingCalls.delete?(aPhoneCall) == nil
    raise 'Only incoming calls can be missed.'
  end
  @missedCalls.add?(aPhoneCall)
end
```

Nous avons également utilisé la méthode `define_singleton_method` pour la classe `Phone` pour redefinir la méthode à adapter. Nous utilisons également d'autre méthode refléxion comme le `send` pour l'exécution d'une méthode. Nous avons du utiliser cette dernière méthode car les méthodes d'`advertise` retourne des blocks.

```
def deploy
    symbol = self.adaptedSelector.to_sym

    res = Phone.send(symbol)
   #"Install adapted method implementation.

    Phone.define_singleton_method(symbol, self.adaptedImplementation)

end
```

5.4 Mixin module

Comme on l'a dit plus haut, notre implémentation utilise le `Mixin module` afin que notre architecture soit simple et facile a comprendre. En effet grace aux `mixins` nous avons pu implémenter des fonctionnalité dans des modules pour ensuite les ajouter à nos classes grace au mot clé `include`. Dans le diagramme ci-dessous on peut voir que lorsqu'on ajoute un module à la classe `Context`, une classe `anonyme` qui contient la référence vers le module s'ajoute dans la hiérarchie d'héritage entre `Context` et son parent :

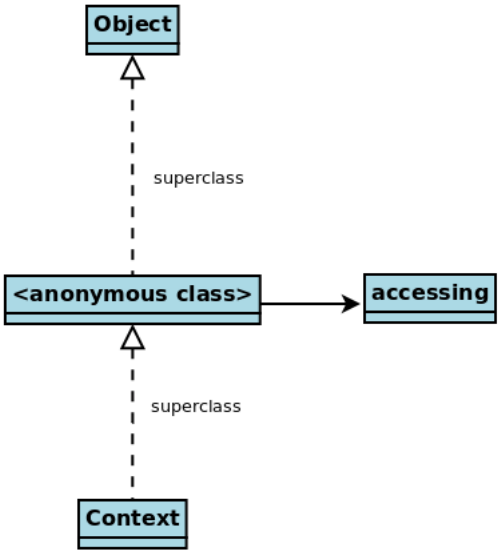

FIGURE 1 – Context inheritance hierarchy

6 Conclusion

Pour conclure nous avons réussit à faire 3 COP sur 4 et avec une solution peut être moins viable pour la 3ème partie du système COP. Malheusement par manque de temps nous avons pas pu terminer une meilleur solution pour la partie 3 ce qui à biensur ampiété sur la bonne réussite de la partie 4. Durant l'implémentation du COP nous avons rencontré quelque difficulé notament lié à la compréhension de pharo mais surtous parceque nous ne pouvions pas faire certaine chose de *Smalltalk* en ruby. Nous avons eu des difficultés lors de la partie 3 surtous parce que les méthodes ne sont pas des objects comme en *Smalltalk*. Donc, reprendre le même cheminement que pour le tp était donc impossible. Dans l'ensemble nous avons apprécié le projet et la découverte des langages *Smalltalk* et *Ruby*.