Design Pattern *Strategy*

Cet énoncé s'étend sur 2 pages.

1. Design Patterns : Elements of Reusable Object-Oriented Software

Un Design Pattern (DP) ou patron de conception est une description d'une solution éprouvée à un problème rémanent dans un contexte de génie logiciel. C'est un modèle de solution (template) décrivant comment résoudre un problème (architectural) de développement logiciel (souvent orienté-objet). Il montre une structure d'objets et les relations entre eux. Chaque implémentation de patron ne peut pas être identique puisqu'appliquée dans des contextes différents.

Il existe différents types de pattern :

— **Créateurs** : Abstract Factory, Builder, Prototype, Singleton
— **Structurels** : Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
— **Comportementaux** : Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

Dans ce TD, nous allons étudier et mettre en oeuvre le patron *Strategy*[1].

2. Strategy

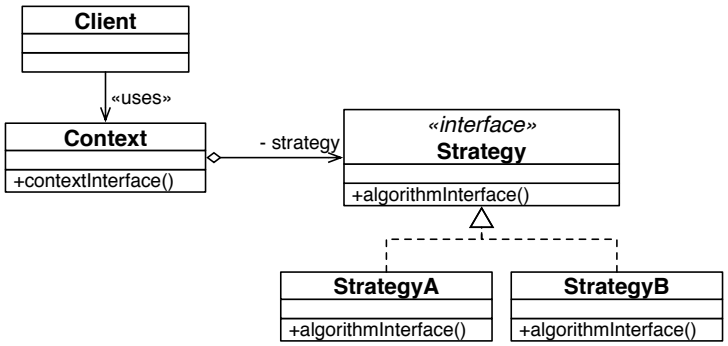

FIGURE 1 – Une description architecturale du DP Strategy

C'est un patron comportemental, dont l'architecture est décrite par le diagramme de la figure 1. Ce patron est utilisé essentiellement lorsqu'il est intéressant d'isoler des variations de comportement d'objets dans des classes différentes, afin d'être en mesure de sélectionner différents algorithmes à l'exécution.

On définit donc une famille d'algorithmes interchangeables, chacun encapsulé dans une classe différente. Par exemple, le comportement d'un robot peut être défini selon différents modes, sélectivement activées en fonction de l'environnement dans lequel ce dernier se trouve (représentation construite à partir des données collectées par les capteurs du robot).

1. Lecture complémentaire : https ://refactoring.guru/design-patterns/strategy

2.1. Implantation

Les classes participantes de ce patron de conception sont :

— **Strategy** : c'est l'interface commune à tous les algorithmes. Elle est utilisée par `Context` pour invoquer un algorithme.

— **StrategyA, StretegyB** : ces classes réalisent les l'interface `Strategy`. Elles implémentent donc chacune un algorithme spécifique.

— **Context** : c'est la classe qui utilise un des algorithmes, grâce à une référence sur le type `Strategy`, et en fonction de la configuration indiquée par `Client`.

— **Client** : c'est la classe qui utilise Context, en le paramétrant pour l'utilisation d'une stratégie adéquate au moment de son instanciation. Généralement, le client passe l'objet représentant la stratégie concrète au contexte, puis le client interagit uniquement avec le contexte.

2.2. Exercice

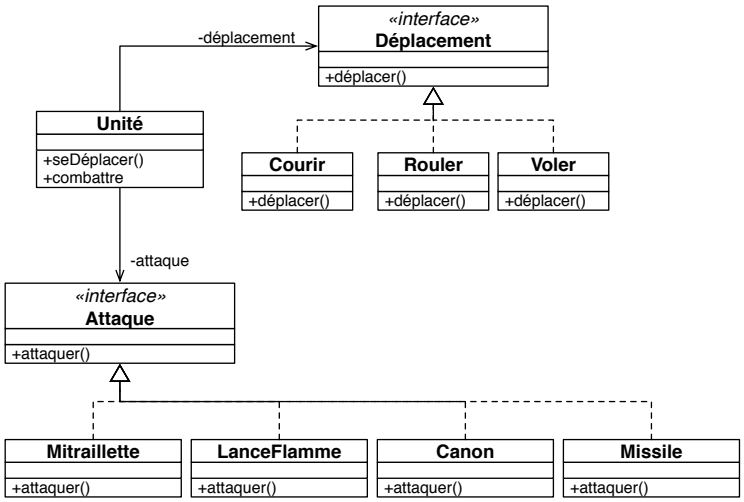

FIGURE 2 – Une instanciation du patron Strategy : jeu de guerre

La figure 2 illustre un modèle de jeu dans lequel une unité peut être un soldat, un char, un hélicoptère, etc. Chaque unité a une capacité de déplacement et de combat spécialisée, variable au sein des unités de même type, par exemple, deux hélicoptères de capacités déplacement et de combat variables. Un soldat peut être équipé d'un lance-flamme, un autre d'une mitraillette, etc.

En utilisant le patron Strategy, écrire un programme du jeu qui met en oeuvre des unités (soldats, chars, hélicoptères) ayant des capacités de déplacement et de combat variables.