Patron de conception et conception logicielle

Ce document à pour but de vous donner quelques information sur les patrons de conception et quand les utiliser. Il est fortement inspiré du site refactoring.guru qui est d'après moi une excellente ressource pour découvrir et comprendre l'intérêt des patrons de conception.

Un patron de conception c'est quoi ?

Un patron de conception (ou *design pattern*) est une solution à un problème récurent en conception logicielle. Mais ce n'est pas une solution clef en main prête à être copiée/collée. C'est un plan à suivre pour mettre en place la solution. Ça demande donc un peu de travail mais permet que la solution proposées soit indépendante du contexte et déclinable à l'infini.

> Les premiers designs patterns ont été formalisé dans le livre Design Patterns – Elements of Reusable Object-Oriented Software de Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides en 1994

Il faut voir un patron de conception comme un patron de couture et la solution qu'il apporte comme un vêtement. Vous pouvez parfaitement trouver des solutions clefs en main sur le web à certains problèmes. Le site stackoverflow est une mine infinie d'information et vous pouvez trouver des solutions à toutes vos interrogations en cherchant bien. Souvent vous aurez un bout de code et une explication assez claire. Mais ces solutions sont un peu difficiles à personnaliser. C'est comme allez acheter un vêtement dans un magasin. Vous avez un vêtement prêt à porter, mais difficilement personnalisable.

À l'inverse vous pouvez acheter un patron pour faire un vêtement. Vous allez avoir un plan des pièces dans différentes tailles et une gamme de montage pour les assembler. Ça demande du travail, les premiers vêtements auront des défauts, mais vous pourrez facilement les personnaliser (tissus, fil, forme etc). Cette solution est l'utilisation d'un patron de conception pour résoudre votre problème. Vous allez adapter le patron à votre cas particulier. Cela fonctionne très bien et est très formateur. La plus grosse difficulté est de savoir quel patron utilisé dans quel cas.

La dernière solution est d'avoir assez de connaissances pour tracer vous même le patron de votre vêtement. Cela demande des compétences en modélisme, de la précision et du temps mais à partir de là vous pouvez théoriquement faire tous les vêtements. En informatique cela revient à avoir assez de compétences en modélisation pour appliquer naturellement les patrons de conception, les "redécouvrir" voir même les fusionner pour répondre à un cas précis.

Un patron de conception pourquoi ?

Tout simplement pour ne réinventer la roue, et avoir une solution **modulable** éprouvée pour répondre à un problème. Beaucoup de patrons de conception peuvent être "redécouvert" une fois la programmation orienté objet et les principes **SOLID** (article wikipedia) maitrisés. Ils utilisent uniquement les principes de bases de la POO (héritage, encapsulation et polymorphisme) pour être indépendant du langage. Leur plus grosse difficulté est qu'ils répondent à des problèmes qu'on ne rencontre que quand on commence à faire des programmes complexes faits pour durer.

Le plus gros apport d'un patron de conception à une solution en dur est la modularité qu'elle va apporter. Le découpage des classes de la solution permet d'avoir un **faible couplage** et une **forte cohérence**. Les classes ne font que peu de tâches et la modification d'une classe doit impacter le moins possible les autres. Ils permettent souvent l'ajout de nouveau code assez à moindre coût. Cela peut sembler inutile dans un projet court, mais dans une application qui va être utiliser pendant des années c'est très important.

En effet une application fait par une entreprise va être utilisée des années et va devoir être mise à jour régulièrement. Si elle est faite comme un monolithe avec beaucoup de code en dur la faire évoluer va être extrêmement difficile. Alors que si elle est codée en utilisant les bons patrons de conception l'évolution sera plus facile.

Les patrons de conception potentiellement utiles pour le projet

Voici une sélection de patrons de conception qui peuvent vous être utile dans le cadre du projet. Lisez-les avec attention et surtout la partie qui vous explique quand les utiliser. Mettre en place un patron de conception pour la première fois n'est pas simple mais montrera votre réflexion sur la conception.

Strategy

Résumé rapide

Pour moi le plus simple des patrons de conception à mettre en place. Le patron *strategy* permet de rendre interchangeable le comportement d'une classe au *runtime* (lors de l'exécution). Il se base sur l'utilisation d'une super classe pour le comportement dont les comportements spécifiques vont hériter.

Quand l'utiliser

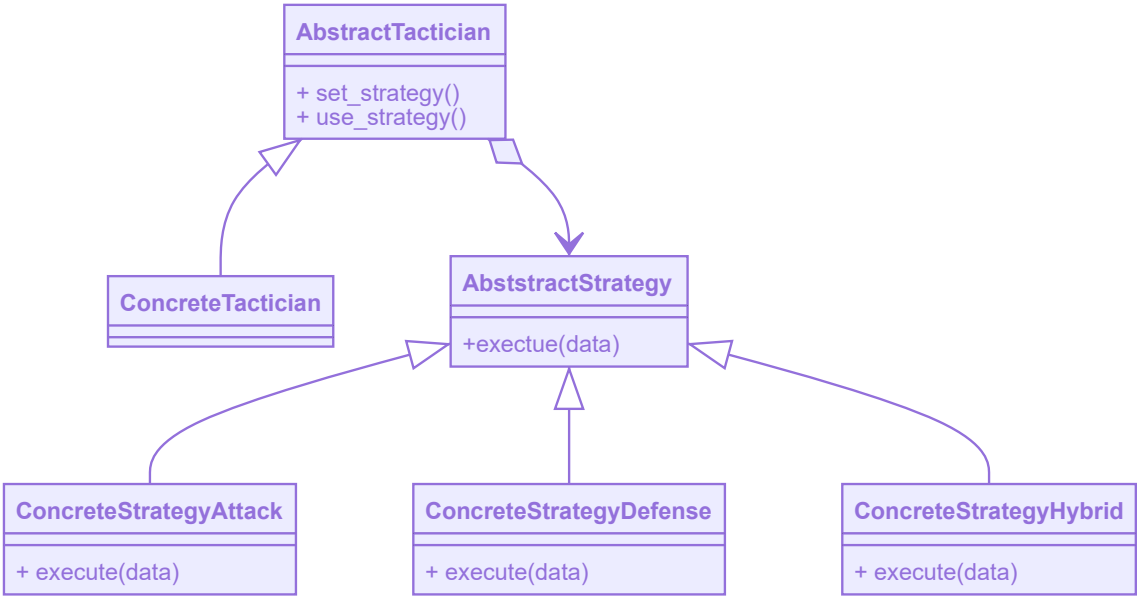
Le patron *strategy* s'adapte bien à des cas de comportements que l'on va devoir changer au *runtime* , ou qui doivent être partagés par différentes classes appartement à la même famille. Si à un moment vous avez du code qui ressemble à

```
def action(self):
    if self.type == "A":
        #code
    if self.type == "B":
        #code
    if self.type == "C":
        #code
    if self.type == "D":
        #code
```

Il y a de forte chance que le patron *strategy* doivent être utilisé. Sinon la question à vous poser est la suivante : est-ce que mon objet à plusieurs comportements possibles que je souhaite définir et changer au *runtime* ? Si oui, il faut utiliser le patron *strategy*

Modélisation

Voici la modélisation simplifiée de ce patron



Les comportements différents sont définis dans les classes `ConcretStrategyAttack`, `ConcretStrategyDefense`, `ConcretStrategyHybrid`. Elles héritent toutes de `AbstractStrategy` comme ça on s'assure qu'elles ont une méthode `execute()`.

La classe que l'on va manipuler dans notre programme et dont le comportement peut changer est la classe `ConcreteTactician` qui hérite de `AbstractTactician`. Cet héritage n'est pas obligatoire si vous avez seulement une seule classe qui voit son comportement varier. Mais une classe abstraite ne coûte pas chère à écrire et permet un code plus souple.

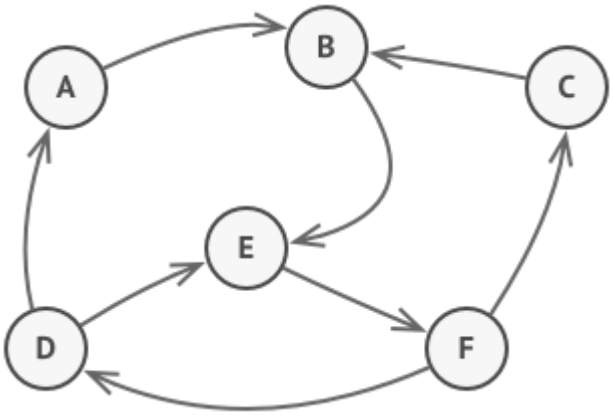
Désormais il nous suffit juste d'appeler la stratégie d'un `ConcretTactician` avec la méthode `use_strategy()` et de la modifier le avec la méthode `set_strategy()` pour avoir un comportement que l'on peut faire évoluer au *runtime*

Pour plus d'info refactor guru strategy

State Machine

Résumé rapide

La patron *State Machine* (ou machine à état) est très proche du patron *strategy*. Le comportement de l'objet va changer en fonction de son état et donc c'est l'état qui va déterminer le comportement et les transitions.



(source refactor guru)

Par exemple si on est dans l'état A et que l'on agit on passe dans l'état B, du B au E etc. On va reprendre le patron *strategy* et ajouter un changement d'état dans les stratégies.

Quand l'utiliser

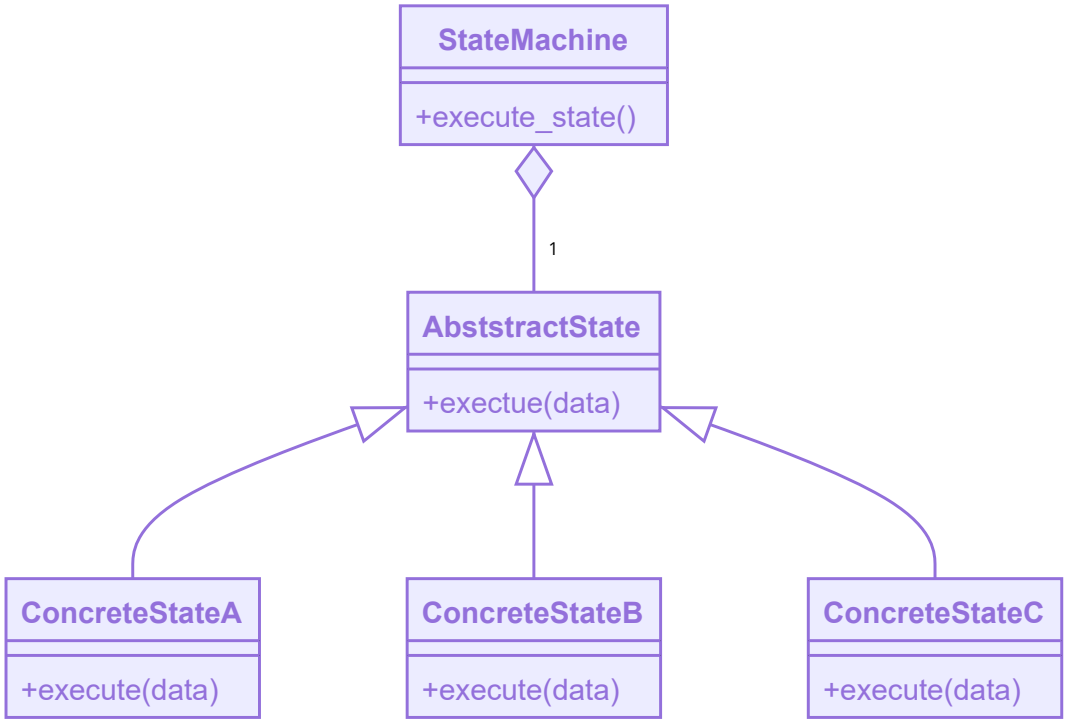
Comme il est très proche du patron *strategy*, *state machine* s'utilise dans des conditions assez similaire. Par exemple si vous avez un code qui ressemble à :

```
def action(self):
    if self.state == "A" :
        #code
        self.state = "B"
    elif self.state == "B" :
        #code
        self.state = "C"
    elif self.state == "C" :
        #code
        if condition :
            self.state = "A"
        else :
            self.state = "B"
```

le *patron machine state* a de fortes chances d'être applicable. Sinon la question à vous poser est la suivante : est-ce que mon objet à plusieurs états, tous avec un comportement différent et des transitions différentes. Si oui, il faut utiliser le patron *machine state*

Modélisation

Voici la modélisation simplifiée de ce patron :



Toutes les comportements des états sont implémentés dans les classes qui héritent de `AbstractState`. La méthode `execute_state()` appelle l'état actuel de la `StateMachine` et réalise son action avec potentiellement un changement d'état à la clef. Pour pouvoir avoir un changement d'état il faut que les états aient accès à l'objet auquel ils sont associés.

Plus d'info [state machine refactor guru]

Template

Résumé rapide

Le patron *template* permet de définir le squelette d'un algorithme dans une classe mère mais laisse les classes filles modifier une partie spécifique sans en changer la structure globale.

Quand l'utiliser

Quand vous avez un algorithme avec une partie fixe et une autre partie que vous voulez pouvoir modifier en fonction du contexte. Par exemple si vous avez du code qui ressemble à :

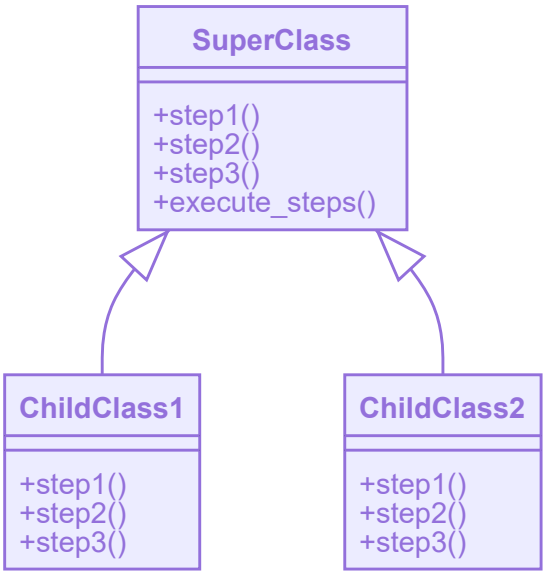
```
def action(self):
    # code fixe d'initialisation

    if self.type = "A":
        # code du type A
    elif self.type = "B":
        # code du type B
    elif self.type = "C":
        # code du type C
    else :
        # code par défaut

    # code fixe de conclusion
```

le patron *template* est fait pour vous.

Vous pouvez vous poser la question suivante pour savoir si vous devez utiliser ce patron : Ai-je un algorithme avec une partie fixe, et une partie qui bouge en fonction du contexte ? Si oui l'algorithme *template* est un bon choix.

Modélisation

Voici la modélisation de se patron.



Dans la `SuperClass` vous allez définir les comportements par défaut des méthodes `step()`, ainsi que l'ordonnancement de ces méthodes dans `execute_steps()`. Chaque classe fille va pouvoir redéfinir ces méthodes si besoin sans touchez à leur ordre exécution.

Plus d'info [template refactor guru]

Bridge

Résumé rapide

Le patron de conception *bridge* est presque comme le patron *strategy* mais plus orienté conception. *Strategy* est une patron de comportement, son but est est de permettre de changer le comportement d'un objet à la volée. *Bridge* quant à lui est un patron de structure. Son but est de permettre une bonne structuration du code et la création d'objet complexe sans pour autant permettre leur changer de comportement au *runtime*. Les buts sont légèrement différents bien que le résultat final soit assez proche.

Ce que va vous permettre le patron bridge est de diviser la structure de vos objets complexes en petits objets que vous allez pouvoir assemblez pour créer votre objet final. Ainsi chaque petit objet ne va gérer que peu de chose et le comportement est bien dissocié.

Quand l'utiliser

Le cas le plus simple est quand vous avez un objet qui semble être en plusieurs parties et que chacune de ces parties varies indépendamment des autres.

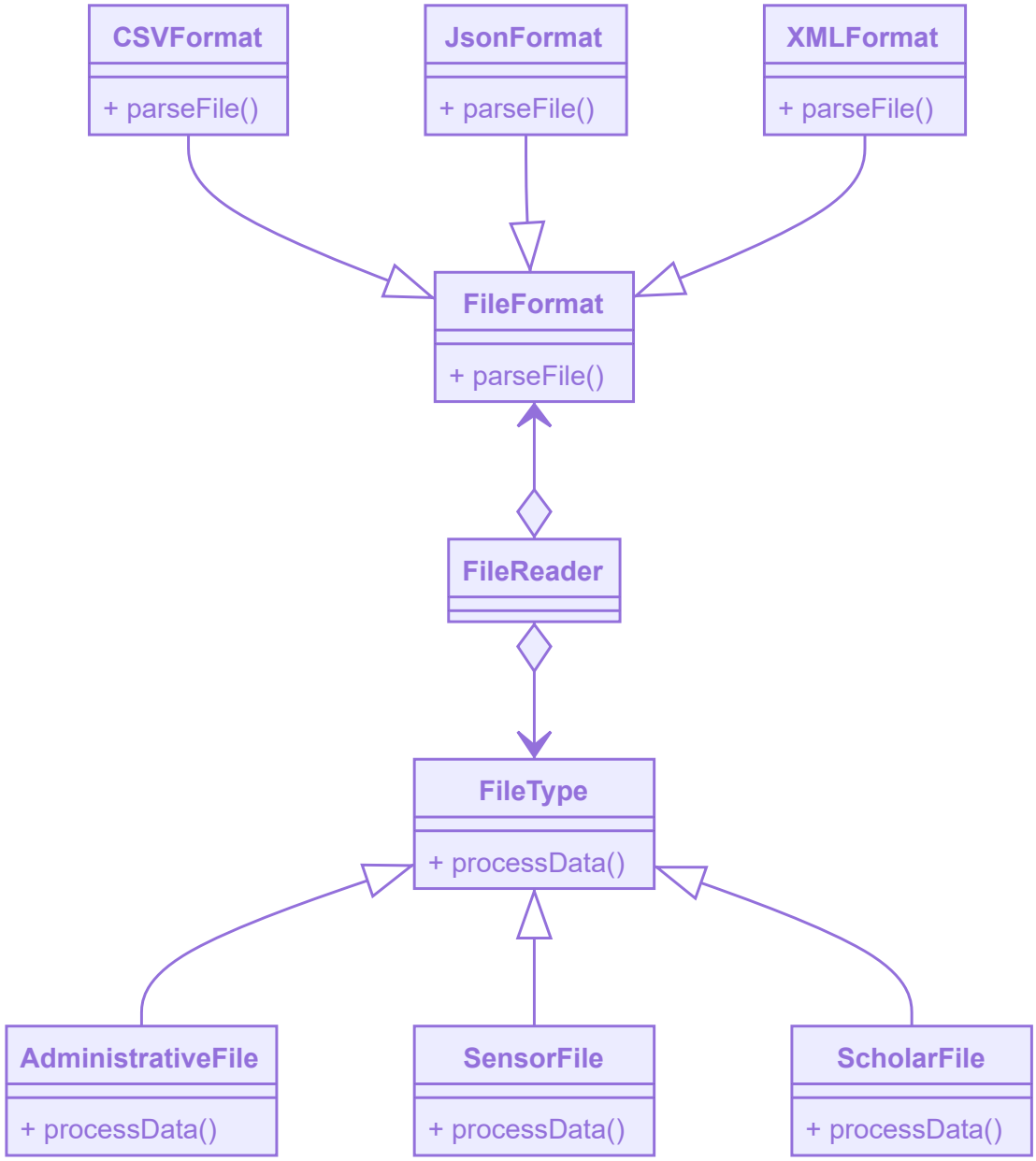
La bonne question à se poser est "Est-ce que j'ai des objets qui sont composés de deux (ou plus) comportements indépendant que je dois assembler et qui me fait exploser mon nombre de classe ?"

Par exemple, imaginez que vous vouliez faire un système qui gère des fichiers de différents formats (json, xml, csv) et avec différents contenu. On peut faire un objet abstrait `AbstractFileReader` et faire des objets qui héritent de lui pour gérer tous nos cas. Avec 3 formats et 4 contenus différent cela fait 12 +1 (pour la classe abstraite) classes à faire, avec beaucoup de redondance de code.

L'autre solution est de dire que l'on va couper notre objet pour gérer d'un coté le type de fichier et de l'autre son contenu. Puis on assemble le tout dans notre classe `FileReader`. Au total cela nous fera 3 + 4 + + 1 = 8 classes. Soit environ 40% de classes en moins. Et si on doit ajouter de nouvelles sources ou de nouveau type c'est très rapide !

Modélisation

Voici une modélisation du patron *bridge* appliqué à l'exemple précédent



Avec cette modélisation on peut facilement faire les 9 croisements possibles sans dupliquer de code.

Pour plus d'information refactor guru bridge

Factory

Résumé rapide

Factory est un patron de création d'objet. Son but est de déléguer la création d'objet à une classe spécialisée. Le plus gros avantage est que si le processus de création de vos objets évolue, comme la *factory* va concentrer une grande partie du comportement de création vous n'allez pas devoir partir à la chasse au code (il risque de falloir aller modifier les appels à la *factory* quand même).

Quand l'utiliser

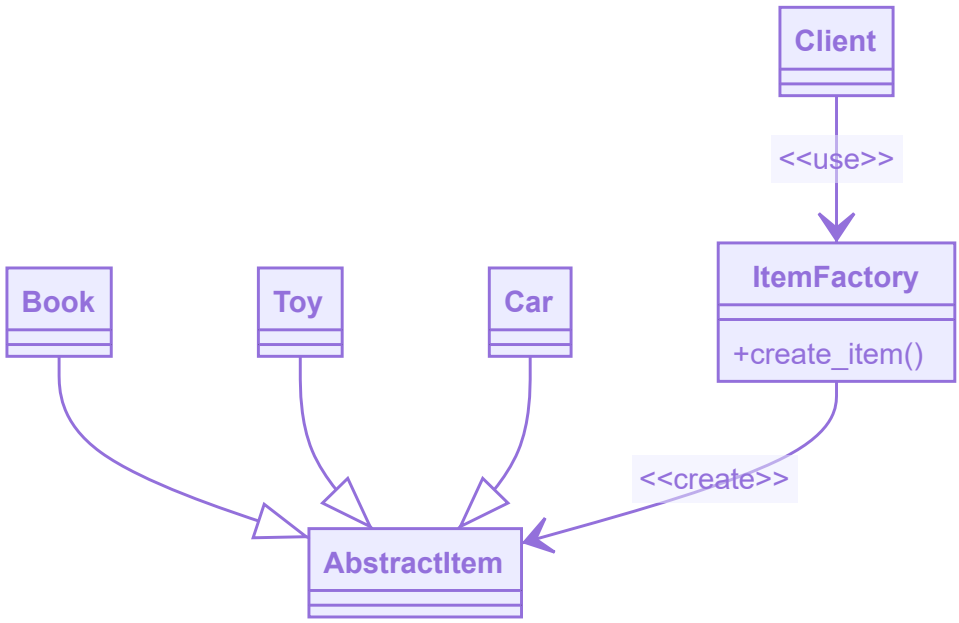
La cas utilisation principal est quand vous avec tout une famille d'objet qui hérite d'une classe mère que vous allez devoir instancier en fonction d'une condition. Par exemple si vous avez du code qui ressemble à cela à plusieurs endroit de votre code :

```python
def create_item(self, type):
    item = None
    if type == "car" :
        item = Car()
    elif type == "book":
        item = Book()
    elif type == "toy":
        item = Toy()
    return item
```

Il est fort possible que vous ayez besoin d'une *factory*.

Modélisation

Voici une modélisation du patron *factory* avec l'exemple ci-dessus



La classe `ItemFacotry` est appelée par une classe cliente pour avoir un objet qui hérite de `AbstractItem`. Ainsi tout le mécanisme de création de l'objet est caché au client. Il fait juste une requête pour avoir l'objet souhaité. Si on veut, on peut même ajouter des méthodes pour produire directement les objets spécifiques.

Pour plus d'information refactor guru factory. La solution proposé est un peu différente de la mienne.

Singleton

Résumé rapide

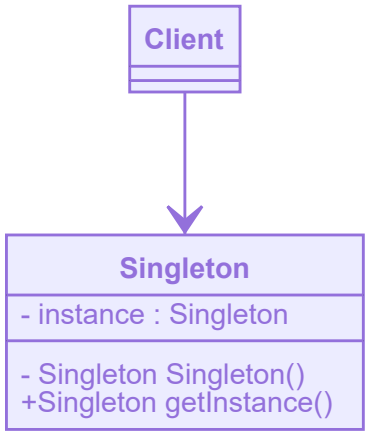
Le patron *singleton* est un patron assez particulier car il permet de s'assurer qu'une objet n'existe qu'une et une seule fois dans toute l'application. Il y a peu de chance que vous l'utilisiez par vous même, mais il est utilisé dans le code fourni pour gérer les connexions alors je fais le point dessus.

Par moment on souhaite avoir un objet unique accessible dans toute l'application. On pourrait faire cela "facilement" en se passante l'objet dans tout le code. Problème, au moindre oubli le code plante. Le plus simple est de s'assurer que quand on instancie un objet de cette classe, on ne crée pas un nouvel objet mais qu'on retourne simplement l'instance déjà existante.

Quand l'utiliser

Quand on veut avoir une objet unique dans tout notre programme. Par exemple un *pool* de connexion, une session avec les données de l'utilisateurs. Cela ne vous arrivera peut-être jamais car c'est un cas assez particulier.

Modélisation



Comme ça ce n'est pas très clair, alors voici le code python associé

```python
class Singleton :
     __instance = None

    def __init__(self) :
        """
        Cette méthode devrait être privée. C'est impossible en python. A la place on         va lever un exception si elle est appellée.
        """
        if Singleton.__instance != None:
            raise Exception("Cette classe est un singleton. Utiliser la "
                            "méthode getInstance()")
        else:
            # Le code pour définir créer l'objet

    @staticmethod
    def getInstance():
        """
        C'est la méthode que l'on va utiliser si l'on veut obtenir l'instance
        du Singleton
        :return: le singleton
        :rtype: Singleton
        """
        if Singleton.__instance is None:
            Singleton()
        return Singleton.__instance


```

Pour plus d'information [code guru singleton]