/*----------------
Design Patterns
Labo 03
Equipe 05 : Loïc Frossard, Valentino Izzo, Luca Meyer, Léon Muller
----------------*/

1. Structure source

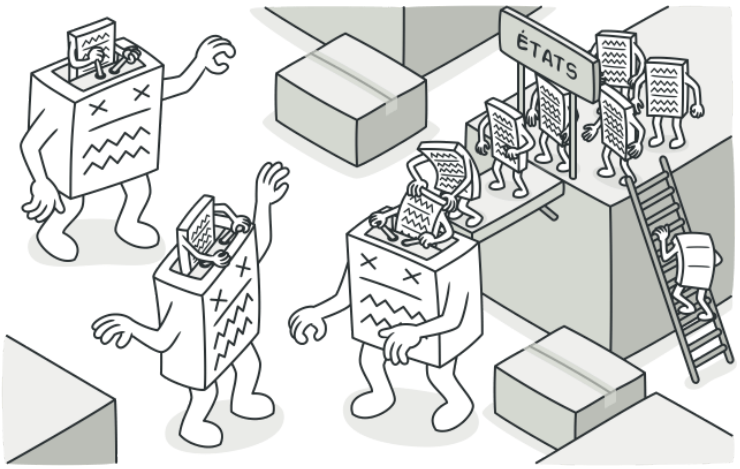
| Decorator
| Composite
| State
|- main.cpp
|- build.sh : Permet de compiler le projet avec g++ depuis un terminal bash ou autre unix
|- labo3.exe : Run le projet
|- Rapport

2. Améliorations possibles

Une amélioration aurait été de mettre plus de temps dans l'infographie afin d'essayer d'obtenir des rendus visuels du décorateur et de l'état.
Nous aurions aussi pu faire une interface afin que l'utilisateur modifie lui-même son panier, et que les fruits changent d'états au cours du temps.

3. Le DP

a. Etat



Le but du pattern état est de modifier le comportement d'un objet lorsque l'état de celui-ci change, il simule donc un changement de classe de cet objet.

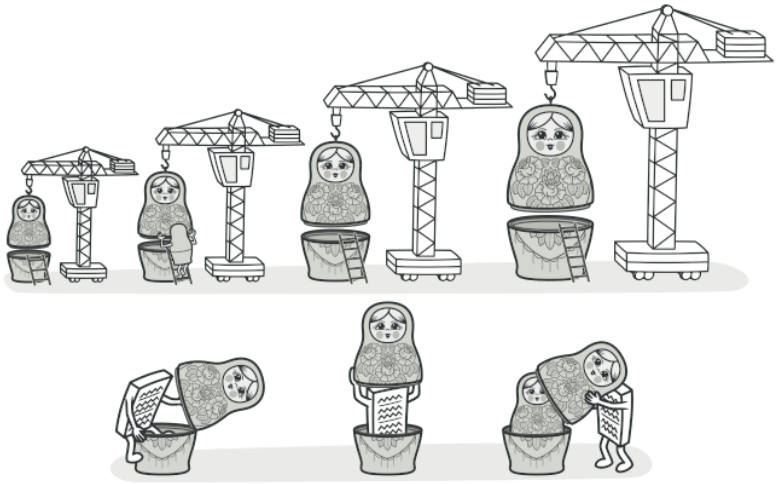
i. Forces

Un des principaux atouts de ce pattern est de pouvoir ajouter des états à nos objets sans pour autant devoir modifier les classes existantes.
Ce pattern nous permet aussi d'éviter de grands switch que l'on aurait dû implémenter afin de gérer toutes les différentes situations.

ii. Faiblesses

Il y a un couplage fort entre l'état et l'objet qui possède un état, ce qui rend difficile l'utilisation d'un état avec d'autres types d'objets s' ils n'ont pas de classe commune.
L'utilisation de ce pattern requiert un nombre important d'états afin d'être vraiment utile, auquel cas il ne fait que compliquer le code sans apporter de simplification majeure.

b. Décorateur



Le but du décorateur est de permettre la modification des comportements (dans notre cas une couleur ou un changement d'affichage) d'un objet en le plaçant dans un nouvel objet (decorator) qui implémente ces changements.

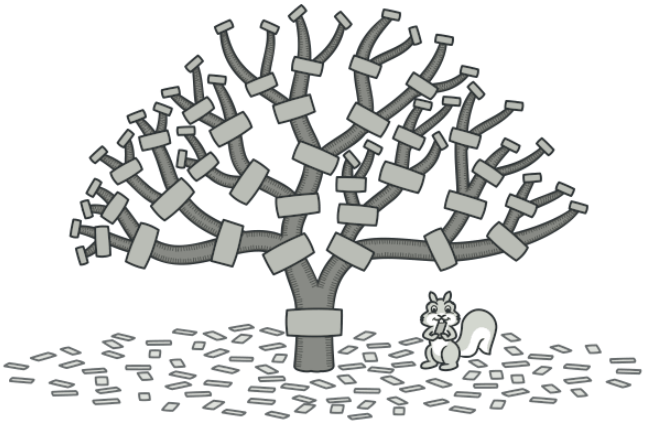
i. Forces

Ce design pattern permet d'ajouter des nouveaux comportements à un objet sans pour autant devoir créer de nouvelles sous-classes ou ajouter une grande quantité de nouvelles méthodes.
Il permet aussi d'assembler/ajouter autant de décorateurs que l'on veut sur l'objet.

ii. Faiblesses

Une faiblesse de ce pattern est qu'un décorateur ne peut pas être retiré (ou en tout cas difficilement) d'un objet une fois qu'il lui est attribué.

c. Composite



Le but du composite est d'avoir une structure en forme d'arbre : chaque instance est représentée par une feuille/branche qui stocke les instances suivantes, que l'on appelle "enfants". On pourrait prendre comme exemple les dossiers et sous-dossiers dans un ordinateur.
Chaque feuille est donc maîtresse d'elle-même et effectue toutes les opérations (d'affichage par exemple) sur elle-même, avant de dire à ses enfants de faire de même (dans le cas d'un appel récursif).

i. Faiblesse

De plus, s'il y a trop de différence entre les membres de l'arbre (cas particuliers), l'algorithme deviendra vite complexe.

ii. Forces

Manipulation d'une même manière toutes les instances de la classe.
On peut lancer une opération sur la racine du composite et la faire se répercuter récursivement sur chacun des enfants jusqu'aux feuilles. Ce pattern est utilisé lorsque l'on agit sur plusieurs objets de la même façon et que l'on se retrouve souvent avec les mêmes implémentations pour chacun d'entre eux.

On peut aussi considérer seulement un petit arbre dans l'arbre total, et donc effectuer des opérations sur des parties distinctes qui composent la structure générale.

4. Conclusion

En implémentant chaque pattern individuellement nous avons réussi à combiner le tout sans couplage fort entre les patterns, ce qui montre leur force et capacité à rendre le code modulable.

Images :

https://refactoring.guru/images/patterns/content/decorator/decorator.png
https://refactoring.guru/images/patterns/content/state/state-fr.png
https://refactoring.guru/images/patterns/content/composite/composite.png