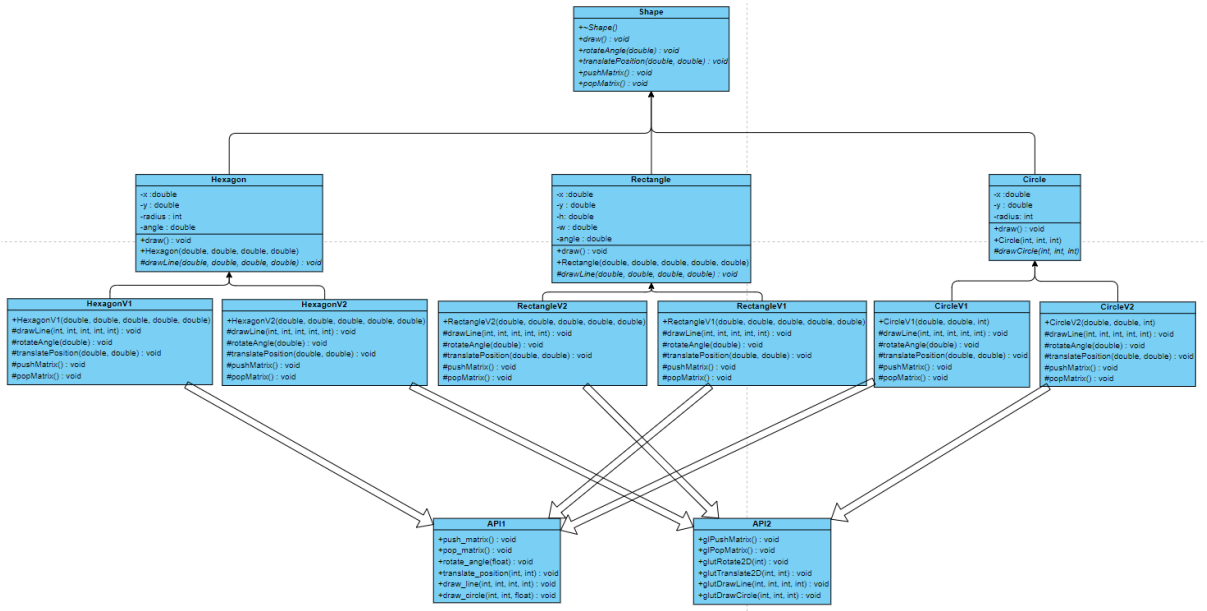
Labo 1

Introduction

Le but principal de ce labo est de nous faire comprendre l'importance de la manière de conception de nos projets.
Ce labo met en valeur les caractéristiques du pattern "Bridge" permettant de remplacer une spécialisation des différentes classes du projet par une agrégation.
Durant ce laboratoire, il nous a été demandé de réaliser la même application consistant à dessiner des formes (cercle, rectangle et hexagone) en utilisant le pattern "Bridge", et en faisant une simple spécialisation des différentes classes.

Analyse

Sans le Bridge



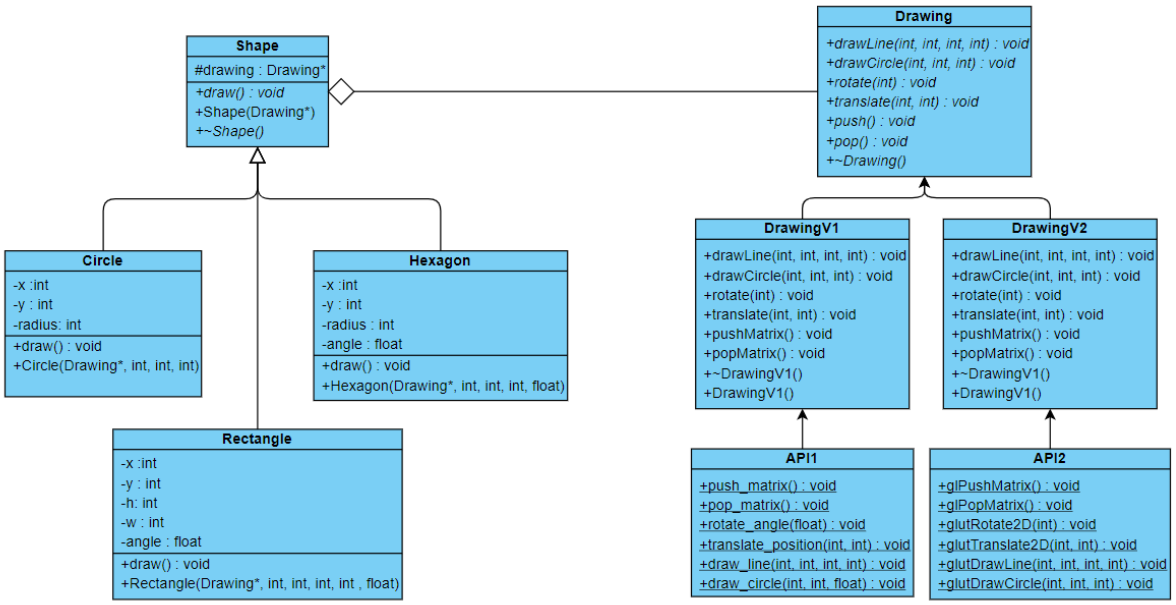
La première intuition que l'on peut avoir pour écrire une bibliothèque qui doit fonctionner avec plusieurs backends/API est de créer une structure avec des classes abstraites, et écrire pour chaque fonctionnalité une classe qui utilise la bonne API.
Le principal inconvénient de cette méthode est qu'il faut écrire un grand nombre de classes, et qu'à chaque fois que l'on souhaiterait ajouter une API, il faudrait à nouveau démultiplier le nombre de classes.

Si par exemple le code à écrire possède 10 classes utiles, il faudra pour chaque API réécrire une classe qui se charge d'appeler les bonnes fonctions de l'API. En ayant 3 backends/API différents, il y aura donc 30 classes d'implémentation, en plus de la spécialisation de nos

différentes classes utiles. De plus, certaines méthodes doivent être implémentées plusieurs fois alors qu'elles font exactement la même chose.

Un autre inconvénient de cette méthode est qu'il est difficile de passer d'une API à une autre puisqu'il faut instancier les classes qui utilisent véritablement une API pour pouvoir les utiliser. Ce qui nous oblige à soit utiliser le pattern de Factory qui va se charger d'instancier les bonnes classes, soit de manuellement indiquer l'API utilisée au moment de l'instanciation de chacun des objets, ce qui est prompt à l'erreur.

Avec le Bridge



Une solution au problème rencontré précédemment est d'utiliser le pattern "Bridge".
Comme avant, on crée notre structure de classes, mais cette fois-ci sans écrire d'implémentation qui utilise une API particulière. A la place, on passe à chacune de nos classes une référence vers une classe abstraite qui va se charger de faire le travail concret en utilisant la bonne API : ce qu'on appelle agrégation.
Nous avons donc transformé l'API en une classe à laquelle on peut accéder sans se soucier de savoir quelle en est l'implémentation. Puisque chaque API est représentée par une classe seule, il n'y a plus besoin de définir chacune de nos classes utiles en fonction de l'API qu'elles utilisent.

Le résultat est que l'ajout d'une API nécessite seulement la création d'une nouvelle classe implémentant les fonctionnalités requises, et le passage d'une API à une autre peut se faire très facilement en instanciant la classe qui utilise l'API que nous souhaitons.

Conclusion

Au travers de ce laboratoire, nous avons pu voir l'importance tout d'abord d'une bonne conception de projet, mais aussi des choix de conception qui permettent ensuite aux développeurs de gagner un temps précieux.
On a pu voir l'efficacité du design "Bridge", le nombre de classes et donc de lignes de codes qu'il a permis de nous faire "économiser". En passant par une conception classique avec une simple spécialisation, la démarche était plus directe et instinctive, mais n'était pas du tout plus efficace, un plus grand nombre de classes ont dû être implémentées.

L'intérêt de ce pattern est de pouvoir passer d'une API à une autre assez aisément sans devoir reprendre l'intégralité du code déjà en place chez un client par exemple.
Il peut être tout aussi utile dans le cas où l'on voudrait réaliser une action commune sur ces différentes formes comme les colorier sans devoir pour chacune d'entre elles, copier coller l'implémentation de ce coloriage.

Au final, le pattern "Bridge" est assez simple à prendre en main, mais le danger est de vouloir l'utiliser dans toutes les situations, alors qu'il pourrait compliquer la situation.