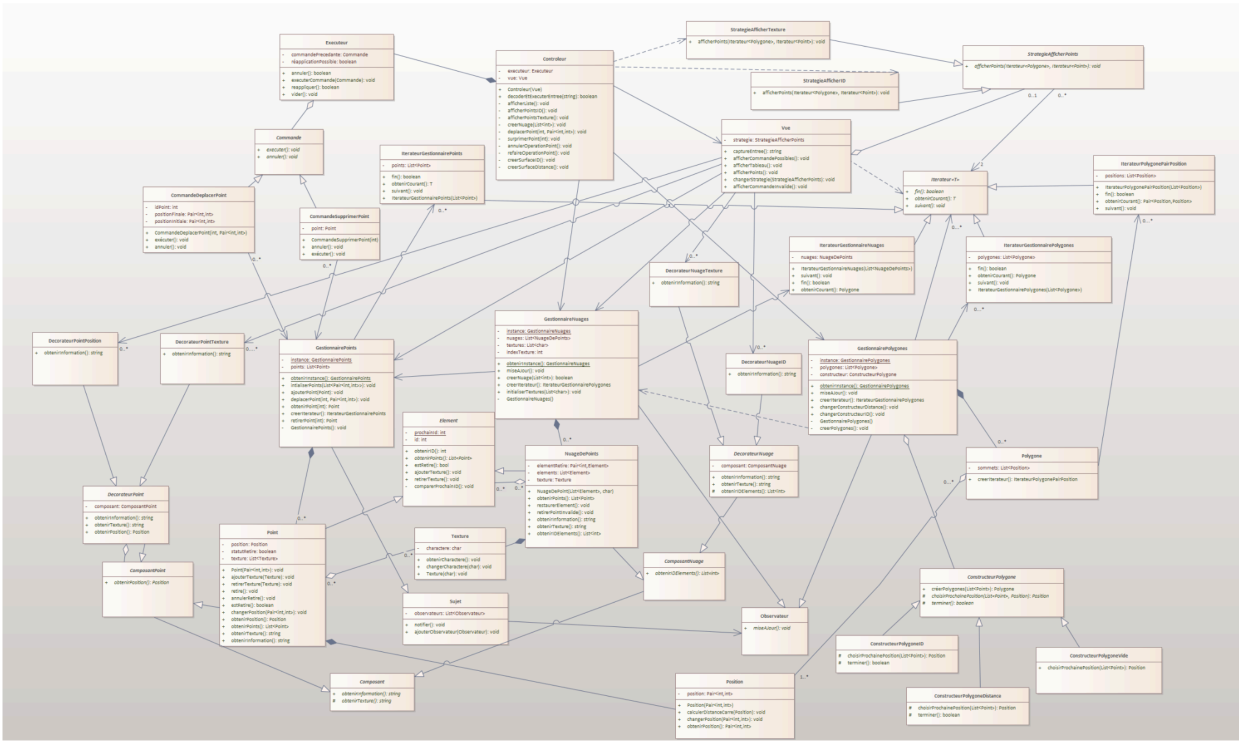
Diagramme de classe



Les multiplicités non-écrites sont des multiplicités de 1.

Présentation de la conception

La plus grande difficulté dans la conception d’un système logiciel est son expansion, ainsi que les modifications qui entraînent de l’usure, ou en anglais du *« technical debt »*. Le système que nous construisons doit, de ce fait, prévoir les avenues possibles d’expansion. Notre conception doit donc être bâtie de manière à ce que les nouvelles fonctionnalités puissent être ajoutées en modifiant le moins de code et de structure possible. Cela a mené à une philosophie de conception axée sur la modularité et visant la séparation explicite des tâches et des données.

Pour séparer les tâches de façon macroscopique, l’architecture classique à trois niveaux et le patron architectural Modèle-Vue-Contrôleur (MVC) nous ont permis de décomposer ce système en niveaux. Du côté de l’architecture à trois niveaux, même si nous n’avons pas implémenté la persistance des données (ce qui peut se faire assez facilement, comme nous le verrons plus loin dans le rapport), nous avons tout de même deux couches explicites, la couche de présentation et la couche de logique de l’application, comprenant les concepts du domaine et les services. Du côté de la présentation, le patron MVC nous permet de définir la vue, responsable de la visualisation du modèle et de la capture des entrées, ainsi que le contrôleur, qui interprète ces entrées et modifie le modèle. Du côté de la logique de l’application, selon le patron MVC, le modèle regroupe toutes les classes qui ne sont ni le contrôleur, ni les commandes, ni la vue, et peut être défini en sous-couches comprenant des services, tels que le gestionnaire de points, ainsi que des concepts du domaine, tel que le point lui-même.

Au niveau de la vue, le patron Stratégie nous a permis de définir plusieurs modes d’affichage lors de la visualisation de la grille de points, en encapsulant chacun des algorithmes représentant ces modes. Dans cette itération du logiciel, on peut choisir d'afficher soit les numéros d'identification, soit la texture de chaque point, selon la stratégie associée à la vue. L'utilisation du patron Stratégie nous permet d'ajouter éventuellement davantage d'algorithmes pour l'affichage des points dans la grille, ainsi que d'offrir plus de contrôle sur l'algorithme que ne le permet le patron Méthode. Par exemple, si l'on voulait un affichage regroupant tous les points dans une liste en bas à gauche, ce patron nous permettrait de le réaliser. De même, si l'on voulait que les textures soient affichées de haut en bas plutôt que de gauche à droite, cela serait possible. Enfin, si l'on voulait afficher une autre donnée, telle que la position, le patron Stratégie nous le permet également.

La vue permet également de formater les données des points et des nuages afin de les afficher dans un tableau. Le patron Décorateur nous permet de réaliser cela, en enveloppant le composant, qu'il s'agisse d'un nuage ou d'un point, d'un décorateur, on peut déterminer dans quel ordre et quelles informations seront fournies lors d'une requête d'information. Si l'on veut changer l'ordre, ou même afficher plus ou moins d'informations, il suffit de choisir d'envelopper ou non notre objet. Ce patron favorise également la maintenabilité, l'ajout de nouvelles informations auxquelles nous devons accéder devient très modulaire et extensible.

Au niveau du contrôleur, deux principaux patrons sont mis en œuvre. Premièrement, le plus évident, le patron GRASP Contrôleur avec la classe Contrôleur, qui permet de gérer les messages d'événements dans une classe centralisée, délégant ainsi les actions concrètes au modèle et séparant nettement la couche de présentation de la couche de logique d'application. Le patron Commande est également très important pour les commandes liées au gestionnaire de points, bien qu'il soit possible de faire en sorte que toutes les commandes modifiant le modèle aient une commande concrète. Le patron Commande permet de définir une classe qui invoque les actions des commandes, ainsi qu'une classe de commande qui offre une interface d'annulation et de réapplication, et un récepteur affecté par les commandes, qui est dans ce cas le gestionnaire de points. Cette conception favorise la ségrégation des tâches entre les deux couches et offre un potentiel d'ajout d'opérations et de commandes simples.

Au niveau du modèle, plus spécifiquement pour les gestionnaires, nous avons décidé d'implémenter le patron Singleton ainsi que le patron Observateur. Premièrement, nous avons appliqué le patron Singleton afin de limiter le nombre d'instances des gestionnaires. Logiquement, il ne peut y avoir qu'un seul gestionnaire dans le contexte du système, puisqu'il existe uniquement pour gérer globalement les instances de points, de nuages de points et de polygones, à l'image de l'exemple classique de la classe Registre. Le patron Observateur permet de notifier automatiquement les observateurs, dans ce cas, le gestionnaire des nuages et le gestionnaire des polygones, lorsqu'un changement survient au niveau du sujet, c'est-à-dire le gestionnaire de points. Ce patron favorise l'extensibilité, puisqu'il évite d'avoir à envoyer manuellement un signal de notification à chacune des classes chaque fois que l'état est modifié. Avec seulement trois gestionnaires, l'absence du patron Observateur ne poserait pas de problème majeur. Toutefois, si l'on souhaite ajouter un autre gestionnaire d'un type différent, par exemple dans le contexte de l'OrthoPrint 3D, un gestionnaire de simulation ou de forces, devoir notifier manuellement tous les gestionnaires augmenterait le risque d'erreurs et d'incohérences entre les éléments du système. L’ajout d’une base de données, comme mentionné précédemment, devient également très faisable grâce à ce patron, la base de données pourrait être notifiée automatiquement chaque fois qu’un élément est modifié au niveau de la logique. L’utilisation du patron Observateur garantit donc une architecture plus robuste, modulaire et extensible, et assure une évolutivité durable du système.

Au niveau des nuages de points, le patron Composite a été utilisé afin de permettre une collection récursive de points et d’autres nuages de points. Ce patron offre une structure hiérarchique uniforme où chaque élément, qu’il s’agisse d’un point individuel ou d’un nuage de points, est traité de manière cohérente. L’utilisation du patron Composite permet ainsi d’implémenter cette organisation de façon élégante.

Le patron Itérateur permet de découpler les objets de la collection qu’ils contiennent, et il est implémenté dans tous les gestionnaires pour lesquels il est nécessaire d’itérer sur leurs éléments, ainsi que sur les arcs du polygone. Ce patron permet de masquer la structure interne de l'agrégation, et assure ainsi une séparation entre la représentation de cette collection et l’utilisation de ses éléments. Il favorise donc l’utilisation d’éléments spécifiques indépendamment de la manière dont la collection est implémentée. Ce patron est particulièrement utile lors de l’implémentation, si l’on doit changer le type de collection, ou même le système complet qui gère la collection d’un type d’objet, l’objet qui itère sur les éléments n’a besoin que de l’interface d’un itérateur abstrait. La modification devient alors simple à réaliser. Le patron Itérateur favorise ainsi la modifiabilité et l’extensibilité.

Au niveau des polygones créés à partir de collections de points fournies par les nuages de points, il est d’abord important de comprendre leur structure. Dans le système, un polygone contient une collection ordonnée de positions représentant les sommets du polygone. Ce format permet une représentation essentielle d’un polygone, qui n’est qu’un cycle dans un graphe complet. Au niveau du gestionnaire des polygones, le patron GRASP Créateur lui associe la responsabilité de créer les polygones. À partir de là, le patron Méthode s’applique à la classe du constructeur de polygones et à toutes ses classes héritières. La classe abstraite ConstructeurPolygones fournit un cadre d’application, une liste d’étapes pour la construction d’un polygone. Dans ce cas, il faut d’abord choisir un premier sommet, puis sélectionner le suivant à partir du dernier de façon itérative. Les classes héritant du constructeur définissent la manière dont le premier point est choisi, ainsi que la logique de sélection des points suivants. Cette approche se prête bien à la modifiabilité, il devient facile de proposer différentes façons de créer des polygones sans perdre l’idée de base, soit la construction d’un cycle. Le patron Méthode constitue donc un ajout positif, permettant la modifiabilité et la mise à l’échelle éventuelle du système.

En somme, l’application des patrons de conception Décorateur, Composite, Modèle-Vue-Contrôleur (MVC), Itérateur, les patrons GRASP d’assignation de responsabilités, Stratégie, Commande, Singleton, Méthode et Observateur assure l’évolutivité du système et réduit l’impact des modifications, ce qui rend l’architecture plus robuste et capable de survivre à l’épreuve du temps.