BORDEAUX INP Ensc

Rapport Technique Gestion d'Expositions et de Vernissages

Mame Awa Ndiaye

1A - Groupe 4

Remis pour le vendredi 23 Mai 2025

Table des matières

1- Introduction

Ce document technique a pour objectif d'expliquer et de justifier le code conçu pour le développement du projet "Gestion d'Expositions et de Vernissages". Le projet a été développé dans l'environnement Visual Studio en utilisant le langage C# sans bibliothèques externes autres que les bibliothèques standards du framework .NET. Dans ce document, nous détaillerons les classes programmées, organisées par rôle, la structure principale du programme, ainsi que les défis rencontrés au cours du développement et les solutions que nous avons mises en place pour les surmonter.

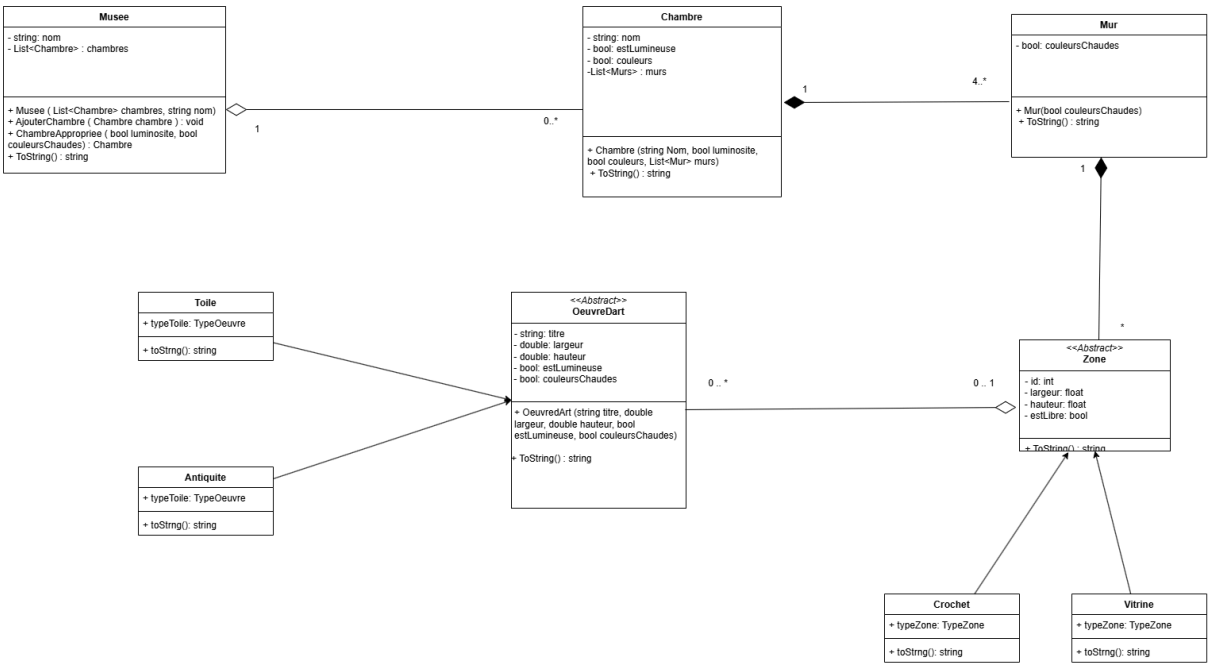
L'objectif du projet est de développer une application permettant à un musée de gérer efficacement ses expositions temporaires et ses vernissages, notamment en optimisant le placement des œuvres d'art selon leurs caractéristiques et celles des salles d'exposition.

2- Choix de la modélisation des données

La modélisation orientée objet a été choisie pour représenter les différentes entités du musée (œuvres d'art, murs, chambres, etc.). Ce choix permet une meilleure organisation du code et une réutilisation des composants, ainsi qu'une représentation plus fidèle des concepts du domaine.

Diagramme de classes UML

Le diagramme ci-dessous représente les relations entre les différentes classes implémentées :



Classe OeuvreDart (abstraite)

Cette classe abstraite sert de base pour toutes les œuvres d'art avec les attributs suivants :

- titre (string) : Le titre de l'œuvre
- largeur (double) : La largeur de l'œuvre en centimètres
- hauteur (double) : La hauteur de l'œuvre en centimètres
- estLumineuse (bool) : Indique si l'œuvre peut supporter une forte luminosité
- couleursChaudes (bool) : Indique si les couleurs dominantes de l'œuvre sont chaudes ou froides

Un enum TypeOeuvreDart définit les types possibles d'œuvres (Toile, Antiquite).

Classe Toile

Cette classe hérite de OeuvreDart et représente une œuvre d'art de type toile.

- TypeOeuvreDart : Définit le type comme étant Toile

Classe Antiquite

Cette classe hérite également de OeuvreDart et représente une œuvre d'art de type antiquite.

- TypeOeuvreDart : Définit le type comme étant Antiquite

Classe Zone (abstraite)

Cette classe abstraite sert de base pour définir les zones d'accrochage sur un mur :

- estLibre (bool) : Indique si la zone est disponible pour accueillir une œuvre
- id (int) : Identifiant unique attribué automatiquement
- largeur (double) : La largeur de la zone en centimètres
- hauteur (double) : La hauteur de la zone en centimètres

Un enum Type Zone définit les types possibles de zones (Toile, Antiquite).

Classe Crochet

Cette classe hérite de Zone et représente une zone adaptée pour accrocher des toiles.

- TypeZone : Définit le type comme étant Toile

Classe Vitrine

Cette classe hérite de Zone et représente une zone adaptée pour présenter des antiquités.

- TypeZone : Définit le type comme étant Antiquite

Classe Mur

Cette classe représente un mur d'une salle d'exposition avec les attributs suivants :

- couleursChaudes (bool) : Indique si la couleur du mur est chaude ou froide
- zones (List<Zone>) : Liste des zones disponibles sur le mur

Cette classe implémente également deux méthodes pour trouver une zone appropriée pour une œuvre :

- zoneAppropriee : Basée uniquement sur les dimensions
- zoneApproprieeBis : Tenant compte du type d'œuvre et du type de zone

Classe Chambre

Cette classe représente une salle d'exposition avec les attributs suivants :

- nom (string) : Le nom de la chambre
- estLumineuse (bool) : Indique si la chambre bénéficie d'une forte luminosité
- couleurs (bool) : Indique si la chambre a des couleurs dominantes chaudes ou froides
- murs (List<Mur>) : Une liste de murs qui composent la chambre

Classe Musee

Cette classe représente un musée contenant plusieurs chambres d'exposition avec les attributs suivants :

- nom (string) : Le nom du musée
- chambres (List<Chambre>) : Une liste de chambres qui composent le musée

Cette classe est responsable de la gestion globale des expositions et du placement des œuvres.

2.1- Structuration du code

Dans le but de respecter le principe de modularité, nous avons implémenté plusieurs classes distinctes, chacune avec ses propres responsabilités :

a. Classe OeuvreDart (abstraite)

La classe OeuvreDart implémente les fonctionnalités suivantes :

- Propriétés d'accès pour tous les attributs avec validation (notamment pour s'assurer que largeur et hauteur sont positives)
- Constructeur de base appelé par les classes qui implémentent cette classe
- Méthode ToString() pour afficher les informations de l'œuvre
- Représente le concept abstrait d'une œuvre d'art, ne pouvant être instanciée directement

b. Classes Toile et Antiquite

Ces classes héritent d'OeuvreDart et ajoutent :

- Un attribut TypeOeuvreDart spécifique à chaque sous-classe
- Une surcharge de la méthode ToString() pour inclure le type de l'œuvre
- La spécialisation du concept d'œuvre d'art en types concrets

c. Classe Zone (abstraite)

La classe Zone implémente les fonctionnalités suivantes :

- Propriétés d'accès pour tous les attributs avec validation
- Un système d'attribution automatique d'identifiants uniques
- Constructeur de base initialisant tous les attributs et marque la zone comme libre
- Représente le concept abstrait d'une zone d'accrochage sur un mur

d. Classes Crochet et Vitrine

Ces classes héritent de Zone et ajoutent :

- Un attribut TypeZone spécifique à chaque sous-classe
- La spécialisation du concept de zone en types concrets adaptés à différentes œuvres

e. Classe Mur

La classe Mur implémente les fonctionnalités suivantes :

- Propriétés d'accès pour tous les attributs
- Constructeur pour initialiser les attributs, avec vérification qu'un mur a au moins une zone
- Deux méthodes de recherche de zone appropriée :
    - zoneAppropriee : vérifie uniquement les dimensions (hauteur et largeur)
    - zoneApproprieeBis : vérifie la compatibilité de type entre l'œuvre et la zone
- Méthode ToString() pour afficher les informations du mur et ses zones

f. Classe Chambre

La classe Chambre implémente les fonctionnalités suivantes :

- Propriétés d'accès pour tous les attributs
- Constructeur pour initialiser tous les attributs, avec vérification qu'une chambre a au moins quatre murs
- Méthode ToString() pour afficher les informations de la chambre

g. Classe Musee

La classe Musee implémente les fonctionnalités suivantes :

- Propriété d'accès pour la liste de chambres
- Constructeur pour initialiser les attributs
- Méthode AjouterChambre pour ajouter une chambre au musée
- Méthode ChambreAppropriée pour rechercher une chambre adaptée à une œuvre en fonction de sa luminosité et de ses couleurs
- Méthode ToString() pour afficher les informations du musée

h. Program.cs

La fonction principale (Program.cs) orchestre l'interaction entre ces différentes classes et teste leurs fonctionnalités :

- Création de plusieurs œuvres d'art (toiles et antiquités)
- Construction d'un musée avec plusieurs chambres
- Recherche de chambres appropriées pour différentes œuvres

- Test des méthodes de recherche de zones appropriées

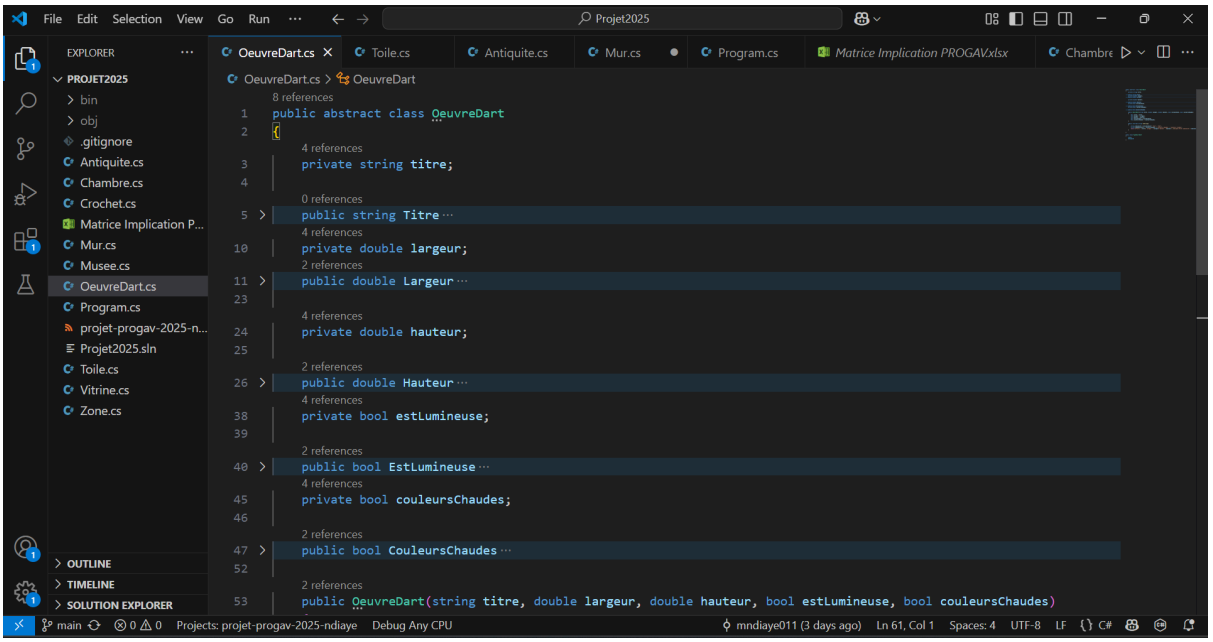
2-2. Implémentation des fonctionnalités avancées

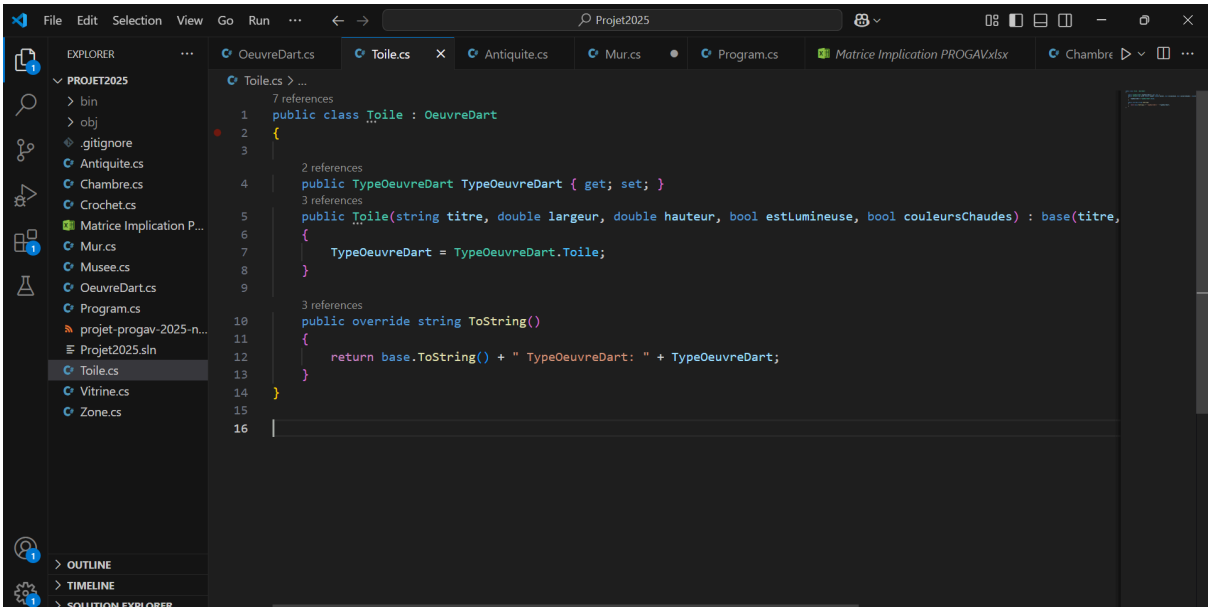
Hiérarchie d'héritage

L'une des principales améliorations apportées au projet depuis la version initiale est l'implémentation d'une hiérarchie d'héritage complète. Cette structure permet de différencier les types d'œuvres d'art et les types de zones tout en partageant les attributs et comportements communs.

Œuvres d'art

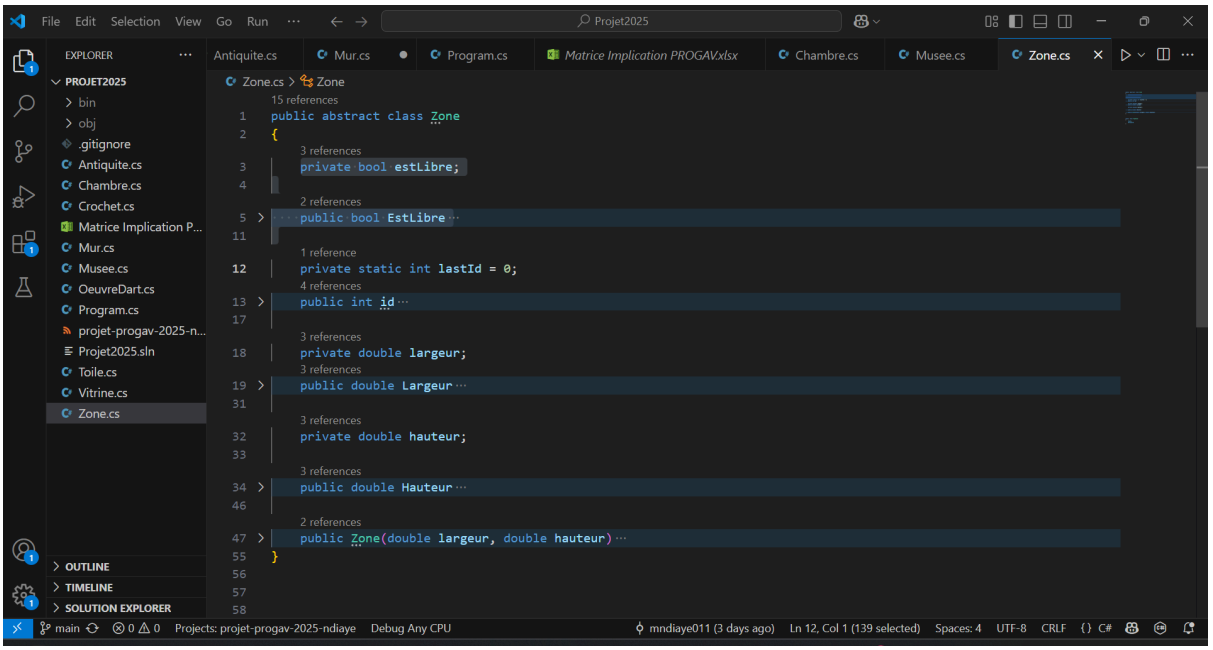
La classe abstraite OeuvreDart définit les propriétés et comportements communs à toutes les œuvres, tandis que les classes dérivées Toile et Antiquite ajoutent leurs spécificités :

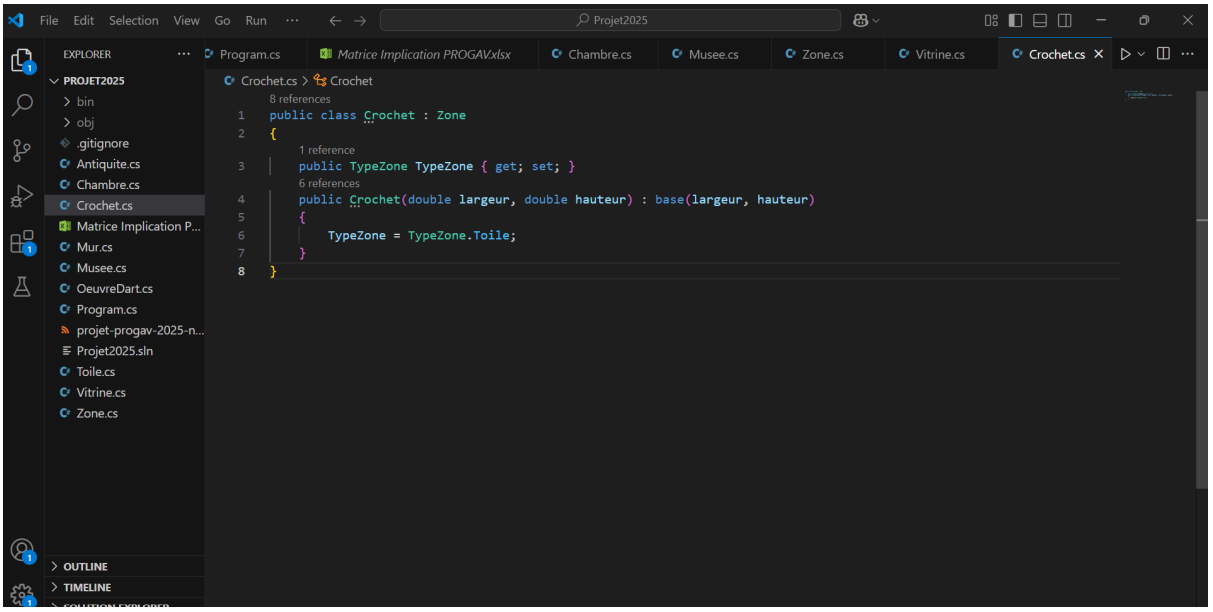




Zones

De même, la classe abstraite Zone définit les propriétés communes, tandis que les classes dérivées Crochet et Vitrine spécialisent les types de zones :





Gestion avancée du placement des œuvres

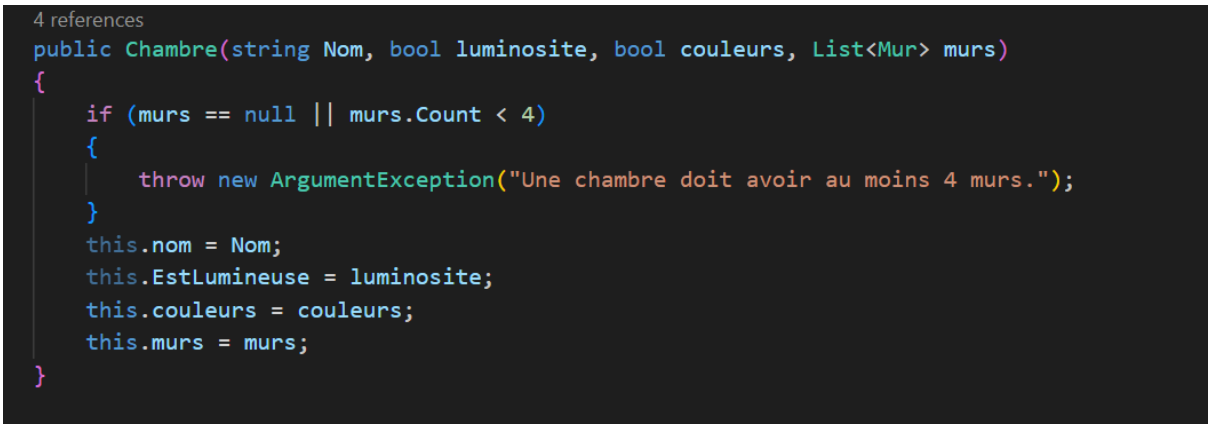
La méthode zoneApproprieeBis de la classe Mur représente une évolution significative par rapport à la méthode zoneAppropriee initiale. Cette nouvelle méthode prend en compte non seulement les dimensions de l'œuvre, mais aussi sa compatibilité de type avec la zone :

Cette implémentation utilise à la fois le polymorphisme (via l'opérateur is) et les règles métier (vérification des dimensions avec une marge de 80cm) pour déterminer la zone appropriée.

Validation des données

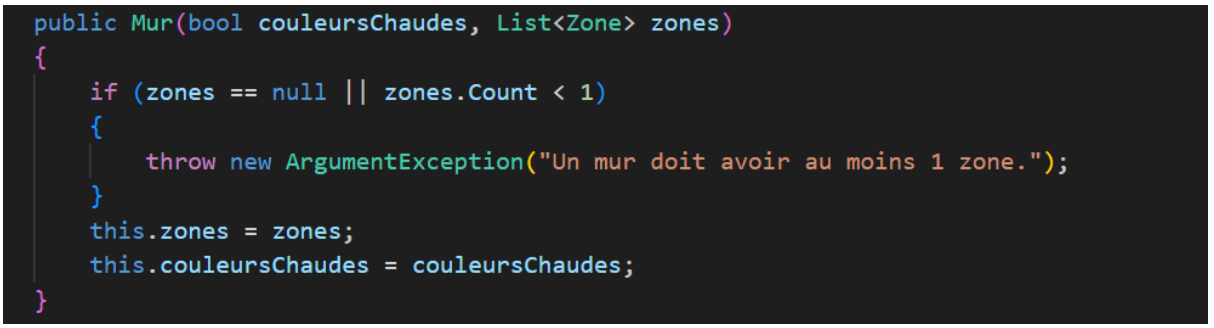
Des validations ont été mises en place pour garantir l'intégrité des données :

1. Vérification qu'une chambre possède au moins 4 murs



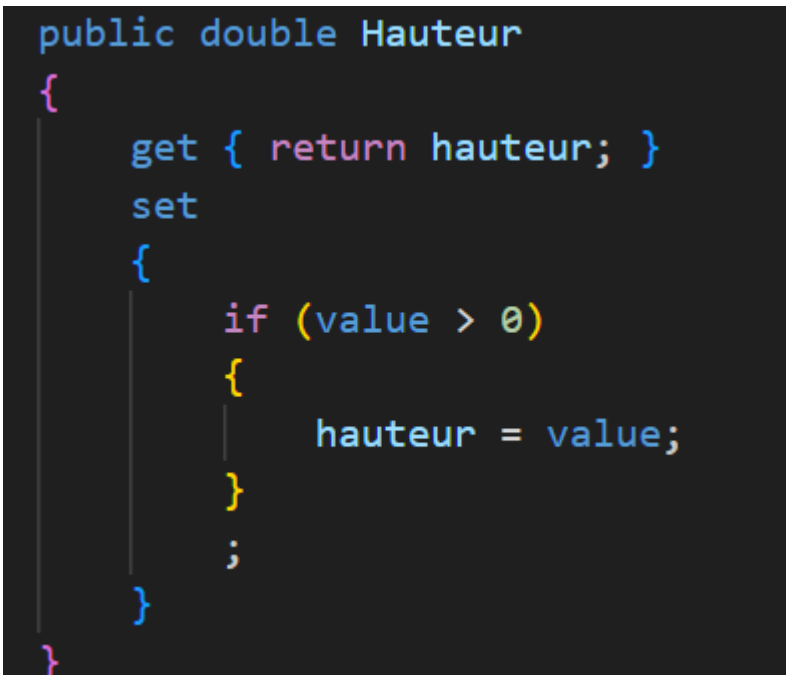
```
4 references
public Chambre(string Nom, bool luminosite, bool couleurs, List<Mur> murs)
{
    if (murs == null || murs.Count < 4)
    {
        throw new ArgumentException("Une chambre doit avoir au moins 4 murs.");
    }
    this.nom = Nom;
    this.EstLumineuse = luminosite;
    this.couleurs = couleurs;
    this.murs = murs;
}
```

2. Vérification qu'un mur possède au moins une zone



```
public Mur(bool couleursChaudes, List<Zone> zones)
{
    if (zones == null || zones.Count < 1)
    {
        throw new ArgumentException("Un mur doit avoir au moins 1 zone.");
    }
    this.zones = zones;
    this.couleursChaudes = couleursChaudes;
}
```

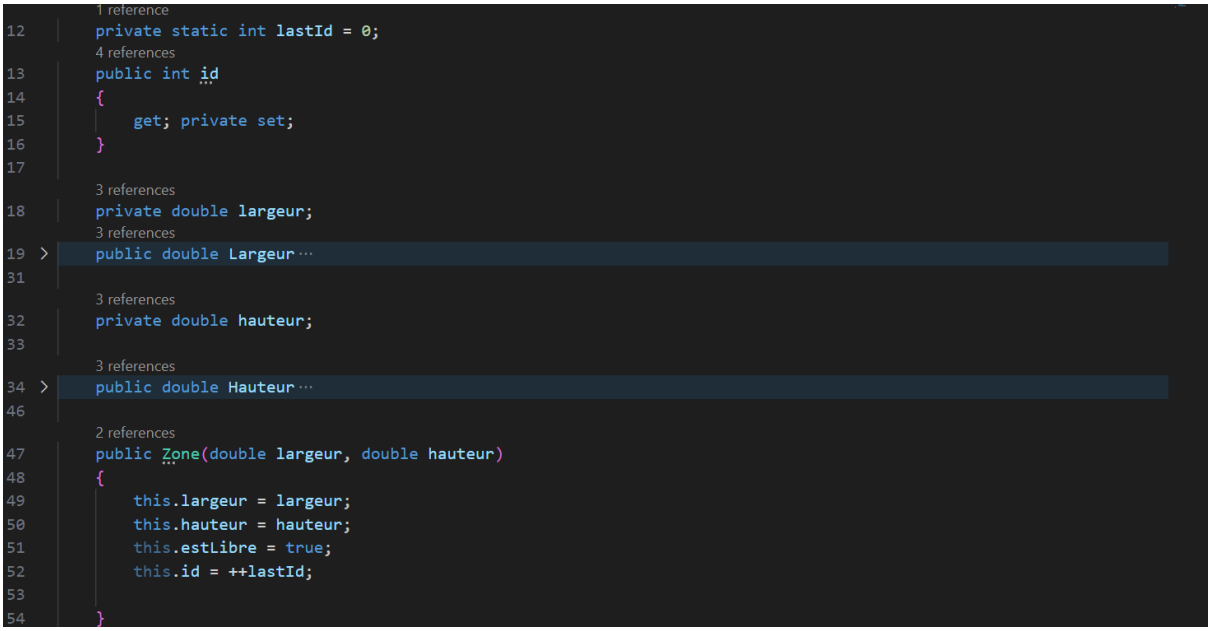
3. Vérification que les dimensions (hauteur et largeur) sont positives



```
public double Hauteur
{
    get { return hauteur; }
    set
    {
        if (value > 0)
        {
            hauteur = value;
        }
        ;
    }
}
```

Système d'identification automatique

Un système d'attribution automatique d'identifiants a été mis en place pour les zones :



```
1 reference
private static int lastId = 0;
4 references
public int id
{
    get; private set;
}

3 references
private double largeur;
3 references
public double Largeur ···

3 references
private double hauteur;

3 references
public double Hauteur ···

2 references
public Zone(double largeur, double hauteur)
{
    this.largeur = largeur;
    this.hauteur = hauteur;
    this.estLibre = true;
    this.id = ++lastId;

}
```

Cette approche garantit que chaque zone reçoit un identifiant unique, simplifiant leur référencement et leur gestion.

3- Tests et validation

Scénarios de test

Le programme principal contient plusieurs scénarios de test pour valider le bon fonctionnement des fonctionnalités implémentées :

1. Création et affichage d'œuvres d'art variées (toiles et antiquités)

```
Toile toile1 = new Toile("MYBLUE", 33.5, 45.0, true, false);
Toile toile2 = new Toile("MYRED", 25.0, 30.0, false, true);
Antiquite antiquite1 = new Antiquite("MYGREEN", 20.0, 25.0, true, true);
Antiquite antiquite2 = new Antiquite("MYYELLOW", 15.0, 20.0, false, false);
```

```
foreach ( var oeuvreDart in oeuvreDarts)
{
    Console.WriteLine($"{oeuvreDart}");
}
```

2. Construction d'un musée avec quatre chambres aux caractéristiques différentes

```
Chambre printemps = new Chambre("Printemps", true, true, new List<Mur> { mur1, mur2, mur3, mur4 });

musee.AjouterChambre(printemps);

Chambre ete = new Chambre("Eté", true, false, new List<Mur> { mur1, mur2, mur3, mur4 });

musee.AjouterChambre(ete);

Chambre automne = new Chambre("Automne", false, true, new List<Mur> { mur1, mur2, mur3, mur4 });

musee.AjouterChambre(automne);

Chambre hiver = new Chambre("Hiver", false, false, new List<Mur> { mur1, mur2, mur3, mur4 });

musee.AjouterChambre(hiver);

Console.WriteLine(musee.ToString());
```

3. Test de la recherche de chambre appropriée

```
// Test de la méthode ChambreAppropriée

Chambre? chambreTrouvee = musee.ChambreAppropriée(toile1.EstLumineuse, toile1.CouleursChaudes);
if (chambreTrouvee != null)
{
    Console.WriteLine($"Chambre trouvée pour la toile 1: {chambreTrouvee.Nom}");
}
else
{
    Console.WriteLine("Aucune chambre trouvée pour la toile 1.");
}


Chambre? chambrePasTrouvee = musee.ChambreAppropriée(toile2.EstLumineuse, toile2.CouleursChaudes);
if (chambrePasTrouvee != null)
{
    Console.WriteLine($"Chambre trouvée pour la toile 2: {chambrePasTrouvee.Nom}");
}
else
{
    Console.WriteLine("Aucune chambre trouvée pour la toile 2.");
}
```

4. Test des méthodes de recherche de zone appropriée

5.
```
// Test de la méthode zoneAppropriee
int zoneTrouvee = mur1.zoneAppropriee(toile1.Hauteur, toile1.Largeur);
Console.WriteLine($"Zone trouvée pour la toile 1: {zoneTrouvee}");

// Test de la méthode zoneApproprieeBis
int zoneTrouveeBis = mur4.zoneApproprieeBis(antiquite2);
Console.WriteLine($"Zone Bis trouvée pour l'antiquité 2: {zoneTrouveeBis}");
```

Résultats des tests

Les tests ont permis de valider :

- La création et l'affichage correct des différentes entités
- Le fonctionnement de la recherche de chambres selon les critères de luminosité et de couleur
- Le fonctionnement de la recherche de zones selon les dimensions
- Le fonctionnement de la recherche de zones selon le type d'œuvre et le type de zone

4- Bilan critique du projet

Au terme de ce projet de développement d'une application de gestion d'expositions et de vernissages, il est important de porter un regard critique sur le travail accompli, d'analyser les réussites comme les difficultés rencontrées, et d'identifier les axes d'amélioration pour de futurs développements.

Ce projet m'a permis de mettre en pratique les concepts fondamentaux de la programmation orientée objet dans un contexte concret. L'architecture développée autour des classes abstraites OeuvreDart et Zone constitue selon moi l'une des principales réussites techniques du projet. L'implémentation de l'héritage avec les sous-classes Toile/Antiquite et Crochet/Vitrine respecte les principes de la POO et offre une structure extensible. Le système d'identification automatique des zones et les mécanismes de validation des données (dimensions positives, nombre minimum de murs et zones) démontrent une approche méthodique de la programmation défensive.

Cependant, je dois reconnaître que plusieurs défis techniques ont émergé au cours du développement. La gestion des listes d'objets et l'implémentation des méthodes de recherche (zoneAppropriee et zoneApproprieeBis) ont nécessité plusieurs itérations avant d'obtenir un fonctionnement satisfaisant. La distinction entre ces deux méthodes, l'une basée uniquement sur les dimensions et l'autre intégrant la compatibilité des types, illustre l'évolution de ma compréhension des besoins métier au fil du projet. De plus, la validation des contraintes (comme s'assurer qu'une chambre possède au moins quatre murs) s'est révélée plus complexe que prévu et a demandé une réflexion approfondie sur la cohérence des données.

L'analyse critique de mon travail révèle néanmoins des limitations importantes qu'il convient d'admettre. Malgré le titre ambitieux "Gestion d'Expositions et de Vernissages", l'application développée ne gère réellement ni les expositions temporaires avec leurs dates et thématiques, ni les événements de vernissage. Cette simplification du domaine métier, bien qu'acceptable dans un contexte pédagogique, crée un décalage entre les ambitions affichées et les

fonctionnalités réellement implémentées. Les critères de placement des œuvres, limités à la luminosité et aux couleurs chaudes ou froides, représentent une vision simplifiée de la réalité muséale qui ne prend pas en compte des aspects cruciaux comme la sécurité, la conservation, ou les contraintes curatoriales.

Sur le plan technique, plusieurs choix de conception mériteraient d'être reconsidérés. L'absence de gestion de la persistance des données limite considérablement l'utilité pratique de l'application, et les algorithmes de recherche linéaire dans les listes pourraient poser des problèmes de performance avec un grand nombre d'œuvres. De plus, la structure actuelle ne permet pas la gestion de conflits lorsque plusieurs œuvres sont appropriées pour une même zone.

Ce projet m'a néanmoins apporté des apprentissages significatifs. Ma compréhension de l'héritage et du polymorphisme s'est considérablement approfondie, particulièrement à travers l'implémentation des classes abstraites et l'utilisation de l'opérateur is pour la vérification des types. La modélisation objet d'un domaine métier complexe m'a sensibilisé à l'importance de bien comprendre les besoins avant de concevoir l'architecture. J'ai également développé mes compétences en matière de validation des données et de gestion des exceptions, même si des améliorations restent nécessaires dans ce domaine.

En conclusion, ce projet représente une étape importante dans mon apprentissage de la programmation orientée objet. Bien que les fonctionnalités développées restent limitées par rapport aux ambitions initiales, la structure mise en place constitue une base solide et extensible. Les limitations identifiées ne diminuent pas la valeur pédagogique de ce travail, mais soulignent plutôt les défis inhérents au développement d'applications métier et l'importance d'une analyse approfondie des besoins avant la conception technique.

5- Conclusion

Ce projet "Gestion d'Expositions et de Vernissages" illustre l'application des principes de la programmation orientée objet pour modéliser et résoudre un problème concret. L'utilisation de l'héritage, du polymorphisme et de l'encapsulation a permis de créer une structure modulaire, extensible et facile à maintenir.

Les principales réalisations incluent :

- Une hiérarchie de classes modélisant fidèlement le domaine métier
- Des mécanismes de validation garantissant l'intégrité des données
- Des algorithmes de placement des œuvres prenant en compte divers critères
- Une structure permettant l'extension future avec de nouvelles fonctionnalités

Bien que certaines limitations existent encore (comme la réutilisation des murs et l'absence de gestion complète de l'attribution des œuvres), la structure actuelle fournit une base solide pour les développements futurs.

Ce projet démontre l'importance d'une conception réfléchie et d'une modélisation soignée pour développer une application complexe de manière incrémentale et maintenable.