
Programmation Avancée
Projet Alternatif 2025 — Gestion d'Expositions
et de Vernissages



Auteur : GARNIER Mathis

2024-2025

Sommaire :

	2
1. Introduction	4
1.1. Contexte du projet	4
1.2. Objectif du jeu	4
2. Choix de modélisation des données	5
2.1. Structure du plateau	5
2.2. Gestion des personnages	5
2.3. Mécanique des grenades et des états du jeu	6
3. Structuration du code	7
3.1. Organisation générale	7
3.2. Fonctionnement du menu principal	7
4.Planning de réalisation	10
4.1. Phases de développement	10
4.2. Gestion des imprévus	11
5. Bilan	12
5.1. Points forts du projet	12
5.2. Axes d'amélioration	12
5.3. Résultats obtenus	13
6. Annexes	14
6.1. Diagramme de structure du plateau	14
6.2. Diagramme de séquence du jeu	14
6.3. Exemples de tests et résultats	15

1. Introduction

1.1. Contexte du projet

Ce projet a été réalisé dans le cadre du module de programmation avancée du semestre 6. Il s'agit d'une simulation de gestion d'un musée, jouable en mode console, codée intégralement en C# sous Visual Studio Code.

L'objectif ici était de construire une application de gestion, mettant en œuvre des règles de compatibilité d'œuvres, de conception orientée objet, et un système de placement intelligent des expositions.

Ce projet nous a permis de mobiliser l'ensemble des notions vues durant l'année : abstraction, héritage, encapsulation, polymorphisme, tout en intégrant des interactions utilisateur robustes et une gestion d'interface ASCII.

1.2. Objectif de l'application




Le but du programme est de permettre à un conservateur virtuel de gérer un musée en y ajoutant des œuvres (toiles ou antiquités), dans des chambres possédant chacune une configuration propre (luminosité, température, capacité des vitrines...).

Chaque œuvre doit être placée en respectant les contraintes suivantes :

- Dimensions compatibles avec la zone murale ou vitrine
- Poids adapté à la vitrine (pour les antiquités)
- Cohérence de lumière (pour les toiles)
- Température des murs (couleurs chaudes ou froides)

Le programme calcule automatiquement les zones compatibles, et propose à l'utilisateur de valider le placement dans la meilleure zone trouvée. Un système d'analyse des raisons de refus est aussi présent, en cas d'échec du placement.

Ce projet met l'accent sur :

- La lisibilité et modularité du code
- Une expérience utilisateur claire et guidée
- La gestion intelligente de contraintes
- L'utilisation avancée de menus, tuples, types abstraits, méthodes virtuelles
- L'affichage dynamique d'un musée en ASCII pour visualiser les zones libres et occupées ( /  , )

J'ai également pu développer de nombreuses compétences en C# à travers ce projet, notamment sur la programmation orientée objet avancée (abstraction, héritage, polymorphisme), la conception de structures de données complexes comme des listes d'objets ou des tuples (Mur, Zone) ainsi que la gestion d'interface console claire et ergonomique (menus, saisies, affichages dynamiques, symboles ASCII).

2. Choix de modélisation des données

2.1. Structure des chambres et zones

La modélisation du musée repose sur une hiérarchie d'objets représentée en programmation orientée objet. Chaque musée contient une liste de chambres, et chaque chambre contient quatre murs.

Les murs sont eux-mêmes composés de plusieurs zones d'exposition, qui peuvent être :

- soit des zones murales (**ZoneMurale**) destinées aux toiles,
- soit des vitrines (**Vitrine**) destinées aux antiquités.

Chaque zone possède une largeur, une hauteur, et une propriété **EstLibre** qui permet de savoir si une œuvre y est déjà placée.

Ce découpage modulaire permet une gestion fine des contraintes d'exposition et un affichage simplifié et clair du musée, à travers une carte ASCII dynamique représentant les zones (■ libres / 🏺, 🖼️ occupées), ce qui renforce l'immersion et la compréhension du système.

2.2. Gestion des œuvres et règles de placement

Deux types d'œuvres peuvent être ajoutés : les toiles et les antiquités, qui héritent toutes deux d'une classe abstraite **Oeuvre**.

Le placement d'une œuvre se fait automatiquement ou manuellement, en vérifiant plusieurs critères :

- Pour une toile :
 - Taille compatible avec une **ZoneMurale**
 - Supporte ou non la forte lumière
 - Esthétique cohérente avec la couleur des murs (chaude ou froide)
- Pour une antiquité :
 - Doit être placée dans une vitrine disponible
 - Le poids ne doit pas dépasser la capacité maximale de la vitrine

Le programme cherche toutes les zones compatibles dans le musée, et :

- sélectionne automatiquement si une seule option est disponible,
- ou affiche un menu de choix à l'utilisateur si plusieurs emplacements sont possibles.

Un système de débogage intelligent (`IsCompatibleDebug`) explique aussi pourquoi une œuvre n'a pas pu être placée (trop grande, mauvaise lumière, vitrines pleines, ...).

2.3. Mécanique d'ajout, affichage et état du musée

Le cœur de l'interaction repose sur une interface console guidée par menu. Le joueur peut :

- ajouter une œuvre via une série de questions (type, dimensions, lumière, couleur, poids...),
- afficher la carte du musée (ASCII, mise à jour en temps réel),
- afficher le détail d'une chambre spécifique,
- ou afficher un récapitulatif complet : nombre total d'œuvres exposées, nombre de zones restantes, chambres encore disponibles.

Chaque ajout est suivi :

- d'un affichage clair de l'état du musée,
- ou d'un rapport d'échec s'il n'a pas été possible de placer l'œuvre.

Les données sont persistantes tant que le programme tourne, avec des structures internes comme des `List<T>` ou des tuples (Mur, Zone). La méthode `AjouterOeuvre()` du musée centralise toute la logique métier, ce qui rend l'application évolutive et facilement testable.

3. Structuration du code

3.1. Organisation générale

Le code a été conçu pour prendre en compte la compréhension et la lisibilité. Chaque fonctionnalité est isolée dans une classe ou une méthode dédiée, afin de faciliter la maintenance et l'évolutivité du projet.

- La classe `Jeu` gère l'interface console et les interactions utilisateur (menu, saisie, affichage).
- La classe `Musee` centralise la logique métier : ajout d'œuvres, affichage global, compatibilité.
- Les classes `Chambre`, `Mur`, `Zone`, `Oeuvre`, `Toile` et `Antiquite` représentent les entités de l'univers du musée avec des responsabilités bien distinctes.

Cette séparation claire permet de faire évoluer indépendamment les modules. Par exemple, modifier les contraintes de compatibilité ou le rendu ASCII ne nécessite pas de modifier l'interface utilisateur.

L'affichage est centralisé dans les méthodes `AfficherCarte()` ou `AfficherChambre()`, ce qui rend le code plus cohérent et réutilisable.

3.2. Fonctionnement du menu principal

Le menu principal, situé dans la méthode `Lancer()` de la classe `Jeu`, agit comme un point d'entrée interactif du programme. Il permet à l'utilisateur de :

1. Ajouter une œuvre
2. Afficher la carte du musée (vue synthétique ASCII)
3. Consulter une chambre spécifique (vue détaillée)
4. Voir un récapitulatif complet du musée
5. Quitter le programme

Chaque choix déclenche une méthode dédiée. Une boucle `while` assure que le menu reste actif tant que l'utilisateur ne décide pas de quitter, rendant l'expérience fluide.

```

public void Lancer()
{
    while (true)
    {
        Console.Clear();
        Console.WriteLine("🎨 Musée Plaisir - Menu principal");
        Console.WriteLine("-----");
        Console.WriteLine("1 - Ajouter une œuvre");
        Console.WriteLine("2 - Voir carte du musée");
        Console.WriteLine("3 - Voir une chambre");
        Console.WriteLine("4 - Voir le récap complet");
        Console.WriteLine("5 - Quitter");
        Console.WriteLine("-----");
        Console.Write("Votre choix : ");
        string choix = Console.ReadLine();

        Console.Clear();

        switch (choix)
        {
            case "1":
                AjouterOeuvre();
                break;

            case "2":
                Console.Clear();
                musee.AfficherCarte();
                Console.WriteLine("\nAppuyez sur une touche pour revenir au menu...");
                Console.ReadKey();
                break;

            case "3":
                Console.Write("Entrez le numéro de la chambre (1 à 4) : ");
                if (int.TryParse(Console.ReadLine(), out int num))
                {
                    Console.Clear();
                    Console.WriteLine($"--- Détail de la chambre #{num} ---\n");
                    Console.WriteLine(musee.AfficherChambre(num));
                }
                else
                {
                    Console.WriteLine("🚫 Numéro invalide.");
                }
                PauseAvantRetour();
                break;

            case "4":
                Console.Clear();
                Console.WriteLine("--- Composition complète du musée ---\n");
                Console.WriteLine(musee.ToString());
                PauseAvantRetour();
                break;

            case "5":
                Console.WriteLine("🎉 Merci d'avoir visité le musée !");
                return;

            default:
                Console.WriteLine("🚫 Choix invalide. Veuillez entrer un chiffre entre 1 et 5.");
                PauseAvantRetour();
                break;
        }
    }
}

```



L'interface est volontairement simple, en mode console, mais enrichie avec des icônes (🎨, 🚫, 🎉, etc.) pour la rendre plus agréable à utiliser.

3.3. Dynamique d'exécution

Le cœur de l'exécution repose sur la gestion intelligente du placement des œuvres, orchestrée par les méthodes `AjouterOeuvre()` dans `Jeu` et `Musee`.

Le programme applique plusieurs étapes :

1. Saisie guidée des propriétés de l'œuvre (type, taille, poids, lumière...)
2. Recherche automatique de zones compatibles en parcourant toutes les chambres, murs et zones
3. Sélection automatique ou manuelle si plusieurs zones sont possibles
4. Affichage du résultat : confirmation ou analyse des raisons de refus

Chaque ajout d'œuvre déclenche un rafraîchissement de la carte ASCII du musée avec les nouvelles zones marquées en rouge ( , ) si occupées.

Le code gère également les saisies invalides et propose à l'utilisateur de réessayer sans provoquer de crash.

Enfin, le récapitulatif global avec `ToString()` fournit un bilan du musée (nombre d'œuvres, chambres pleines ou non, zones libres).

Cette structure rend le programme robuste, intuitif et évolutif, tout en respectant le cahier des charges pédagogique.

```

public bool IsCompatibleDebug(Oeuvre oeuvre)
{
    if (oeuvre is Toile toile)
    {
        bool bonneLumiere = toile.Fortelumiere == (Luminosite > 5);
        bool couleurOK = false;
        bool tailleOK = false;

        foreach (var mur in Murs)
        {
            if (mur.Couleur == (toile.CouleurChaude ? CouleurMur.Chaude : CouleurMur.Froid)
                couleurOK = true;

            foreach (var zone in mur.Zones)
            {
                if (zone is ZoneMurale && zone.EstLibre &&
                    zone.Largeur >= toile.Largeur + 40 &&
                    zone.Hauteur >= toile.Hauteur + 40)
                {
                    tailleOK = true;
                }
            }
        }

        if (!bonneLumiere)
            Console.WriteLine("❌ Lumière incompatible");
        if (!couleurOK)
            Console.WriteLine("❌ Couleur du mur incompatible");
        if (!tailleOK)
            Console.WriteLine("❌ Toile trop grande pour les zones murales");

        return bonneLumiere && couleurOK && tailleOK;
    }

    if (oeuvre is Antiquite antiq)
    {
        bool vitrineCompatible = false;
        bool vitrineExistante = false;
        bool vitrineTropPetite = false;

        foreach (var mur in Murs)
        {
            foreach (var zone in mur.Zones)
            {
                if (zone is Vitrine vitrine)
                {
                    if (zone.EstLibre)
                    {
                        vitrineExistante = true;

                        if (vitrine.CapaciteMax >= antiq.Poids)
                        {
                            vitrineCompatible = true;
                        }
                        else
                        {
                            vitrineTropPetite = true;
                        }
                    }
                }
            }
        }

        if (!vitrineCompatible)
        {
            if (!vitrineExistante)
                Console.WriteLine("❌ Aucune vitrine disponible (occupée ou absente)");
            else if (vitrineTropPetite)
                Console.WriteLine("❌ Trop lourd pour les vitrines de cette chambre");
        }

        return vitrineCompatible;
    }

    Console.WriteLine("❌ Type d'oeuvre non reconnu.");
    return false;
}

```

`IsCompatibleDebug()` analyse en détail pourquoi une œuvre ne peut pas être placée dans une chambre, en expliquant chaque contrainte non satisfaite (taille, lumière, couleur, poids, etc.).




4. Planning de réalisation

4.1. Phases de développement

Le développement du projet s'est déroulé en plusieurs étapes structurées.

La première phase a été celle de l'analyse et de la conception. Il s'agissait de bien comprendre les contraintes du projet : exposition de différentes œuvres selon leurs caractéristiques, gestion des chambres et des zones d'affichage, et affichage dynamique d'un musée en console. J'ai commencé par modéliser les entités principales du projet : `Oeuvre`, `Zone`, `Mur`, `Chambre`, `Musee`, en adoptant une approche orientée objet avec des classes bien séparées.

Une fois la structure définie, j'ai progressivement implémenté les fonctionnalités essentielles. J'ai commencé par l'ajout des œuvres et la gestion des zones compatibles, en introduisant des règles métiers spécifiques (compatibilité lumière, couleur, dimensions, poids, etc.). J'ai ensuite créé la logique de recherche automatique d'une zone compatible dans toutes les chambres, avec une option de sélection manuelle si plusieurs choix étaient possibles.

Par la suite, j'ai mis en place une interface console ergonomique : un menu principal clair, des messages d'erreur, une pause entre les actions et un affichage visuel stylisé à l'aide de symboles (, , ).



Une attention particulière a été portée à la méthode `IsCompatibleDebug()`, qui explique précisément pourquoi une œuvre ne peut pas être placée. Cela a été utile à la fois pour l'utilisateur, mais aussi pour moi en phase de test.

Ensuite, j'ai développé la visualisation du musée sous forme ASCII, ligne par ligne, en représentant chaque chambre et ses zones. Cette partie a été affinée pour que l'affichage réagisse correctement à chaque ajout d'œuvre (zones devenant rouges automatiquement).

Enfin, la dernière phase a été dédiée au débogage, à la sécurisation des saisies utilisateur (`TryParse`, relance en cas d'erreur), à l'amélioration de l'expérience utilisateur (menu clair, retours visuels, résumé final du musée), et à la mise au propre du code (commentaires, renommage, suppression des duplications).

4.2. Gestion des imprévus

Le projet a rencontré plusieurs imprévus techniques au cours du développement.

Un premier souci est apparu lors de la mise à jour de l'affichage ASCII après l'ajout d'une œuvre. Bien que la zone soit occupée en mémoire, l'affichage ne reflétait pas toujours correctement l'état (, ). Après analyse, cela venait de la logique de récupération des zones pour l'affichage, qui listait parfois des doublons ou ignorait les bonnes zones. J'ai refait entièrement la méthode `GetSymbolesZones()` pour régler ce problème.

Un autre imprévu a concerné la gestion des erreurs de saisie. Initialement, le programme crashait si l'utilisateur entrait une lettre au lieu d'un nombre. J'ai corrigé cela en encapsulant toutes les entrées dans des boucles avec `int.TryParse()` pour garantir une saisie propre et robuste.

L'implémentation de `IsCompatibleDebug()` a également été délicate : je voulais qu'elle affiche des messages clairs mais concis (par exemple "trop lourd" ou "taille insuffisante") sans surcharger l'utilisateur avec des blocs de texte. Il a fallu équilibrer l'aspect informatif avec la lisibilité.

Enfin, le fait de gérer plusieurs types d'œuvres avec des règles différentes a exigé un bon usage du polymorphisme (avec `abstract`, `override`, etc.), ce qui m'a amené à repenser l'organisation de certaines classes (`Zone`, `ZoneMurale`, `Vitrine`).

Malgré ces imprévus, j'ai pu les résoudre un à un grâce à une approche itérative, en m'appuyant sur des tests fréquents, une architecture modulaire et des messages de débogage bien pensés.

5. Bilan

5.1. Points forts du projet

L'un des principaux points forts de ce projet réside dans la structure modulaire et la lisibilité du code. Chaque action, comme l'ajout d'une œuvre, la sélection automatique d'une zone compatible, ou encore l'affichage du musée, est isolée dans une méthode spécifique. Cela permet une grande clarté dans l'organisation et facilite la maintenance.

J'ai également respecté les bonnes pratiques de codage, en appliquant des conventions comme `PascalCase` et `camelCase`, et en ajoutant des commentaires utiles dans les zones complexes du code. Les règles de compatibilité ont été implémentées de façon réaliste, avec des contraintes sur la lumière, la température des murs, la capacité des vitrines ou la taille des œuvres.

Un autre aspect marquant est l'interactivité du programme. Le menu en console est clair et intuitif, les erreurs de saisie sont gérées proprement, et le retour visuel lors de l'ajout d'une œuvre (avec des 🍷 , 🖼️ ou vertes 🟢) rend l'expérience utilisateur fluide et agréable.

La méthode `IsCompatibleDebug()` est un bel exemple d'outil de validation pédagogique, permettant d'expliquer pourquoi une œuvre ne peut pas être placée. Elle a été précieuse autant pour le débogage que pour l'utilisateur.

Enfin, ce projet m'a permis de consolider mes compétences en C# : objets, héritage, abstraction (`Zone`, `ZoneMurale`, `Vitrine`), gestion des listes, saisie sécurisée (`TryParse`), ainsi que l'usage d'affichage ASCII dynamique et de menus interactifs en console.

5.2. Axes d'amélioration

Même si le projet est fonctionnel et fidèle au cahier des charges, plusieurs améliorations restent envisageables.

Tout d'abord, l'affichage ASCII, bien que stylisé, pourrait être encore plus lisible et évolutif : des bordures plus cohérentes, un centrage automatique ou une vue isométrique pourraient améliorer le rendu visuel du musée.

Du côté du code, certaines parties comme l'ajout d'une œuvre ou la recherche de zone compatible pourraient encore être factorisées pour éviter la duplication (ex. logique partagée entre Toile et Antiquite). Certaines conditions ou méthodes longues gagneraient à être découpées davantage.

Le projet pourrait également intégrer un système de sauvegarde, une gestion des suppressions, ou un historique des œuvres ajoutées.

Sur le plan UX, un tutoriel intégré ou des messages guidés pourraient aider les nouveaux utilisateurs à comprendre les règles de placement et à éviter les erreurs.

Enfin, je pourrais intégrer un système de test plus systématique pour valider tous les cas d'ajout, les limites de compatibilité, ou les zones pleines, afin d'assurer une fiabilité maximale dans toutes les configurations.

5.3. Résultats obtenus

Le projet est globalement fonctionnel et conforme aux objectifs initiaux. L'utilisateur peut ajouter différentes œuvres au musée, qui sont automatiquement placées selon des contraintes réalistes (dimensions, lumière, couleur du mur, poids, etc.). Le menu console est clair, la navigation fluide, et les messages d'erreur ou de confirmation sont bien gérés. L'affichage des chambres et de la carte réagit correctement à l'ajout des œuvres.

Cependant, tout n'est pas parfait. L'affichage ASCII, bien qu'efficace pour une visualisation rapide, reste sommaire. La disposition des zones peut parfois manquer de clarté visuelle, notamment lorsqu'on ajoute plusieurs œuvres : certaines zones apparaissent en 🏺, 🖼️, mais il peut y avoir des incohérences selon la logique d'affichage ou l'ordre des murs parcourus. C'est un point que j'aurais pu retravailler davantage.

En termes de validation, j'aurais aussi pu ajouter davantage de tests visuels dans le code : par exemple, remplir volontairement toutes les zones d'une chambre pour voir comment le système réagit, ou tester différents scénarios de placement impossibles pour valider la robustesse du `IsCompatibleDebug()`. Cela aurait renforcé la fiabilité globale de mon programme.

Sur le plan technique, j'ai toutefois acquis une solide maîtrise des concepts orientés objet, et j'ai su intégrer des règles complexes dans un projet interactif et évolutif. Ce travail m'a donc permis de

progresser significativement et de mieux comprendre la logique de conception d'une application structurée en C#.

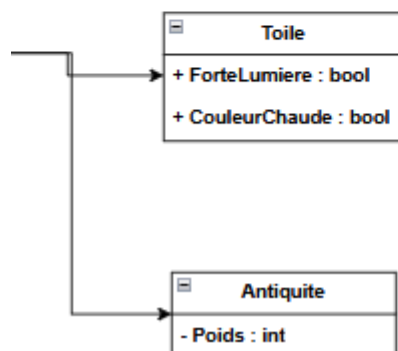
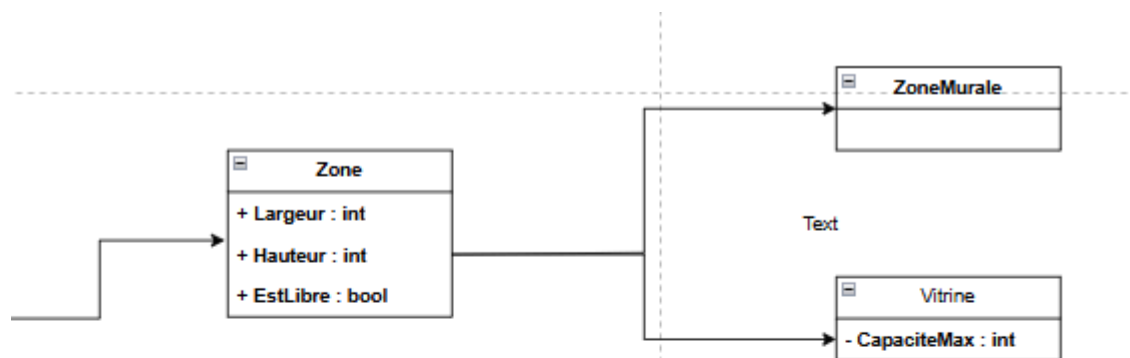
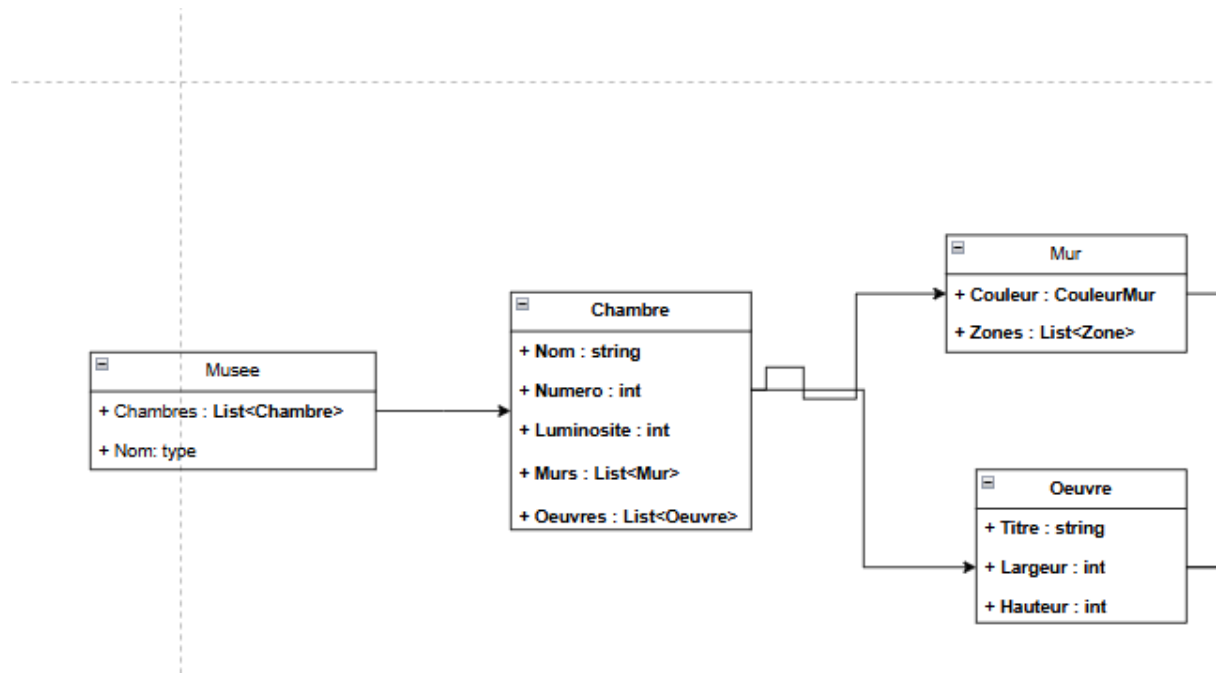
6. Annexes

6.1. Modélisation objet (Diagramme de classes UML)

L'application repose sur une modélisation orientée objet claire et extensible. Voici les principales classes du projet et leurs relations (tu peux insérer ici un diagramme UML réalisé avec un outil comme draw.io, StarUML ou Lucidchart) :

- Musée
 - Contient une liste de `Chambre`.
 - Gère l'ajout d'œuvres, l'affichage global, les statistiques.
- Chambre
 - Composée de 4 `Mur`, chacun ayant une couleur (chaude ou froide).
 - Contient une liste d'œuvres exposées.
- Mur
 - Possède une couleur (`CouleurMur`).
 - Contient une liste de `Zone`.
- Zone (classe abstraite)
 - Propriétés : largeur, hauteur, booléen `EstLibre`.
 - Deux classes dérivées :
 - `ZoneMurale` : pour les toiles.
 - `Vitrine` : pour les antiquités (avec `CapaciteMax`).
- Oeuvre (classe abstraite)
 - Deux sous-classes :
 - `Toile` (lumière, chaleur, taille).
 - `Antiquite` (taille, poids).

Toutes les classes sont modulaires, et certaines contiennent des méthodes d'analyse ou de validation comme `IsCompatible()` ou `IsCompatibleDebug()`.



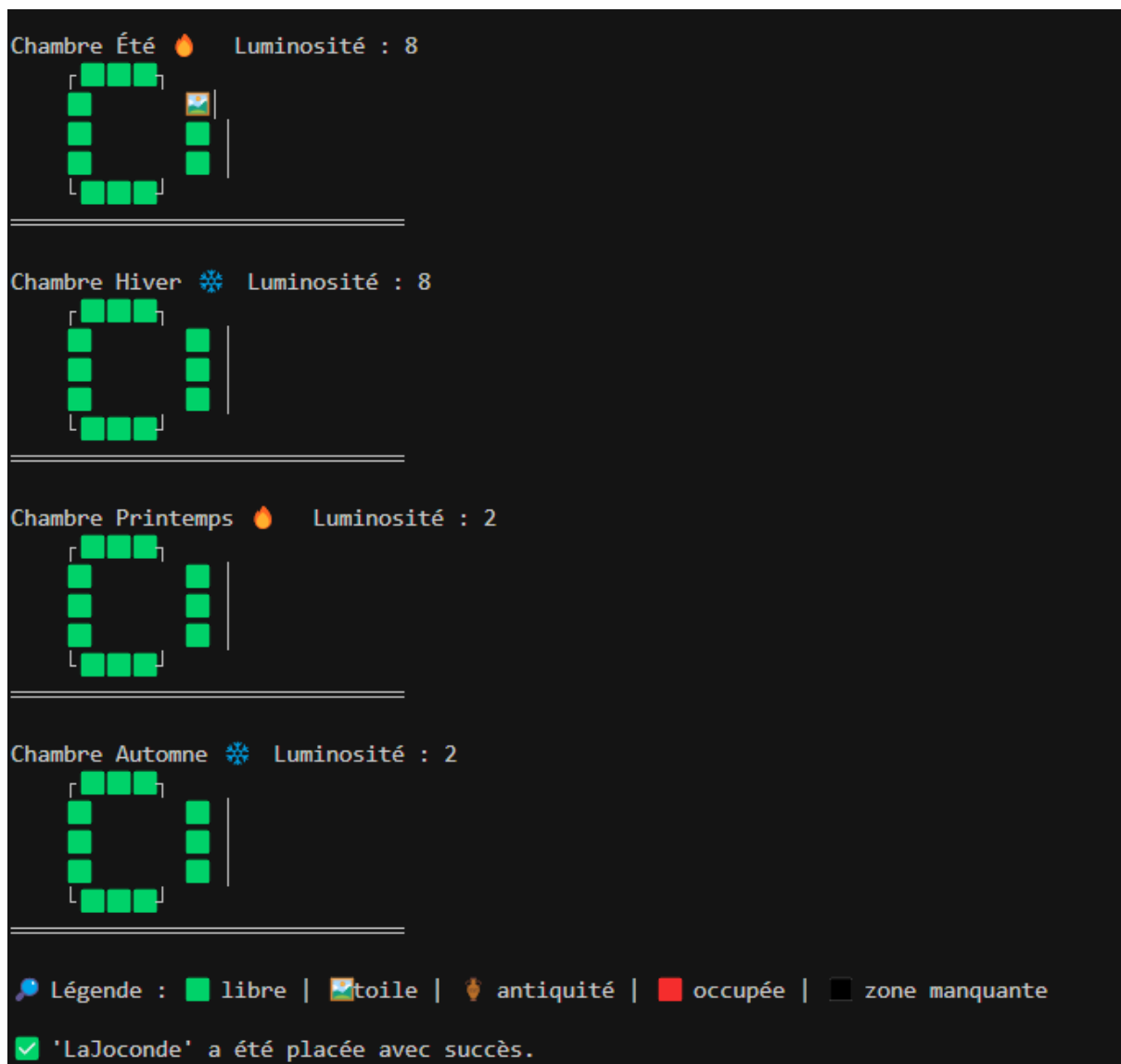
6.2. Tests réalisés

Test 1 : Placement automatique d'une œuvre

Scénario : L'utilisateur ajoute une toile compatible avec la chambre 1.

Résultat attendu : L'œuvre est automatiquement placée dans une zone murale libre.

Résultat obtenu : L'œuvre apparaît dans la chambre, la zone devient 🖼️ dans la carte.



Test 2 : Message d'erreur si l'œuvre ne peut pas être placée

Scénario : L'utilisateur tente d'ajouter une antiquité trop lourde.

Résultat attendu : Message d'erreur clair, `IsCompatibleDebug()` identifie la contrainte.

Résultat obtenu : Message affiché ("trop lourd pour les vitrines de cette chambre").

```
❌ Impossible de placer 'Zeus'.
🔍 Analyse des raisons :

🔵 Chambre « Été » (#1)
❌ Trop lourd pour les vitrines de cette chambre

🔵 Chambre « Hiver » (#2)
❌ Trop lourd pour les vitrines de cette chambre

🔵 Chambre « Printemps » (#3)
❌ Trop lourd pour les vitrines de cette chambre

🔵 Chambre « Automne » (#4)
❌ Trop lourd pour les vitrines de cette chambre
```

Test 3 : Multiples zones compatibles

Scénario : Une œuvre correspond à plusieurs zones disponibles.

Résultat attendu : L'utilisateur peut choisir manuellement.

Résultat obtenu : Une liste s'affiche avec les options disponibles.

```
+ Ajouter une œuvre au musée
-----
Quel type d'œuvre ? (1 = Toile, 2 = Antiquité)
1
Titre de l'œuvre : LaJoconde
Largeur : 20
Hauteur : 20
Supporte la lumière forte ? (o/n) : o
Couleurs chaudes ? (o/n) : n

🧐 Plusieurs zones compatibles trouvées. Veuillez choisir :

[0] 🖼️ ZoneMurale sur mur 1, zone 1 (150x120) | chambre #2
[1] 🖼️ ZoneMurale sur mur 1, zone 2 (140x110) | chambre #2
[2] 🖼️ ZoneMurale sur mur 2, zone 1 (150x120) | chambre #2
[3] 🖼️ ZoneMurale sur mur 2, zone 2 (140x110) | chambre #2
[4] 🖼️ ZoneMurale sur mur 3, zone 1 (150x120) | chambre #2
[5] 🖼️ ZoneMurale sur mur 3, zone 2 (140x110) | chambre #2
[6] 🖼️ ZoneMurale sur mur 4, zone 1 (150x120) | chambre #2
[7] 🖼️ ZoneMurale sur mur 4, zone 2 (140x110) | chambre #2

Votre choix (entrez un numéro valide) : █
```

Test 5 : Résumé complet du musée

Scénario : Plusieurs Œuvres sont placées.

Résultat attendu : ToString() affiche toutes les Œuvres, les zones libres, etc.

Résultat obtenu : Données exactes, statistiques affichées correctement.

```
🎨 Bienvenue au Musée Plaisir !

=====
🏠 Chambre « Été » (#1) | Luminosité : 8 | Murs : 🔥 Chauds
🧱 Mur Chaude avec 3 zones :
- 🖼️ ZoneMurale (150x120) → 🟢 libre
- 🖼️ ZoneMurale (140x110) → 🟢 libre
- 💎 Vitrine (90x90) → 🟢 libre | capacité max : 100kg
🧱 Mur Chaude avec 3 zones :
- 🖼️ ZoneMurale (150x120) → 🟢 libre
- 🖼️ ZoneMurale (140x110) → 🔴 occupée
- 💎 Vitrine (90x90) → 🟢 libre | capacité max : 100kg
🧱 Mur Chaude avec 3 zones :
- 🖼️ ZoneMurale (150x120) → 🔴 occupée
- 🖼️ ZoneMurale (140x110) → 🟢 libre
- 💎 Vitrine (90x90) → 🟢 libre | capacité max : 100kg
🧱 Mur Chaude avec 3 zones :
- 🖼️ ZoneMurale (150x120) → 🟢 libre
- 🖼️ ZoneMurale (140x110) → 🟢 libre
- 💎 Vitrine (90x90) → 🟢 libre | capacité max : 100kg
🖼️ Œuvres exposées :
🖼️ Toile : Nuit Etoilée | Taille : 20x20 | Exposition à la lumière : oui | Couleur : chaude
🖼️ Toile : 10 | Taille : 10x20 | Exposition à la lumière : oui | Couleur : chaude
-----
🏠 Chambre « Hiver » (#2) | Luminosité : 8 | Murs : ❄️ Froids
🧱 Mur Froide avec 3 zones :
- 🖼️ ZoneMurale (150x120) → 🟢 libre
- 🖼️ ZoneMurale (140x110) → 🟢 libre
- 💎 Vitrine (90x90) → 🟢 libre | capacité max : 600kg
🧱 Mur Froide avec 3 zones :
- 🖼️ ZoneMurale (150x120) → 🟢 libre
- 🖼️ ZoneMurale (140x110) → 🔴 occupée
- 💎 Vitrine (90x90) → 🟢 libre | capacité max : 600kg
🧱 Mur Froide avec 3 zones :
- 🖼️ ZoneMurale (150x120) → 🟢 libre
- 🖼️ ZoneMurale (140x110) → 🟢 libre
- 💎 Vitrine (90x90) → 🟢 libre | capacité max : 600kg
🧱 Mur Froide avec 3 zones :
- 🖼️ ZoneMurale (150x120) → 🟢 libre
- 🖼️ ZoneMurale (140x110) → 🟢 libre
- 💎 Vitrine (90x90) → 🟢 libre | capacité max : 600kg
🖼️ Œuvres exposées :
🖼️ Toile : LaJoconde | Taille : 20x20 | Exposition à la lumière : oui | Couleur : froide
-----
```

7. Conclusion

Ce projet de simulation de musée m'a permis de mettre en œuvre l'ensemble des compétences acquises au cours de ma formation en programmation orientée objet en C#. J'ai pu consolider ma maîtrise des concepts fondamentaux comme l'héritage, la composition, les listes, la gestion des objets, mais aussi découvrir des aspects plus poussés comme la gestion des interactions dynamiques via la console, l'implémentation de contraintes logiques, ou encore la vérification de compatibilité entre objets.

La construction de ce projet a exigé une réelle rigueur dans la structuration du code, la modularisation des fonctionnalités et le respect des conventions de nommage. J'ai également appris à intégrer une logique de test et de débogage continue, pour m'assurer du bon fonctionnement des mécaniques du jeu (placements, affichages, compatibilité des œuvres...).

Même si le projet est fonctionnel et atteignait les objectifs fixés, certaines limites d'ergonomie et de lisibilité (comme l'affichage console ou les tests visuels) pourraient encore être optimisées. De nouvelles fonctionnalités (comme un système de sauvegarde, une interface graphique ou des stats détaillées) pourraient aussi enrichir l'expérience.

En résumé, ce travail représente une étape marquante dans mon parcours d'apprentissage. Il m'a offert une expérience complète de développement, depuis la modélisation objet jusqu'à la finalisation d'un programme interactif, tout en me préparant à des projets plus ambitieux, individuels ou en équipe, avec une meilleure maîtrise de l'environnement C#.

Je finirai ce rapport par des remerciements pour la chance que vous m'avez donnée afin que je puisse réaliser ce projet.