UFR MATHÉMATIQUES, INFORMATIQUE, MÉCANIQUE ET AUTOMATIQUE

Design Patterns : peut-on aller encore plus loin ?

Rapport d'Initiation à la Recherche

Date de rendu :

Lundi 6 mai 2024

Auteurs :

Alexandre UNTEREINER

Vincent LE

Professeur référent :

Dominique MICHEL

Établissement : Université de Lorraine - UFR MIM

Formation : 1re année de Master en Informatique

Année universitaire : 2023 - 2024

Table des matières

Résumé

Ce rapport examine l'intégration et le potentiel des Design Patterns dans la programmation orientée objet. Comment ces modèles conceptuels, déjà présents dans des langages comme C++ et Java, pourraient être davantage intégrés pour optimiser et simplifier le développement logiciel. Le document explore l'histoire des Design Patterns, leurs avantages, notamment en termes de réutilisation et de maintenabilité du code, ainsi que les défis associés aux anti-patterns. En se projetant vers l'avenir, le rapport envisage l'évolution des Design Patterns face aux technologies émergentes et aux exigences modernes du développement. Une étude de cas sur le projet "AngryBalls" illustre l'utilisation pratique du modèle Entité-Composant-Système (ECS) pour dépasser les limites des modèles traditionnels. Cette analyse approfondie vise à contribuer à l'amélioration continue des pratiques de programmation et à l'adoption de solutions de conception plus efficaces et adaptatives.

1 Introduction

1.1 Présentation du sujet

Les Design Patterns (DP), éléments fondamentaux de la programmation orientée objet, sont intégrés dans de nombreux langages tels que C++, Java, Python, etc... où ils prennent la forme d'interfaces, de classes abstraites, et d'autres structures. Ces motifs de conception jouent un rôle crucial en fournissant des solutions éprouvées aux problèmes récurrents de développement logiciel. Cependant, malgré leur prévalence, la question demeure : existe-t-il d'autres DP universels et bénéfiques qui mériteraient une intégration directe dans les langages orientés objet ?

1.2 Objectifs du rapport

L'enjeu de ce rapport est double : d'une part, il vise à enrichir le corpus théorique des Design Patterns en proposant éventuellement de nouveaux candidats à l'intégration dans les langages de programmation. D'autre part, il cherche à offrir aux concepteurs de langages orientés objet des outils de compréhensions approfondies pour évaluer quels DP pourraient être internalisés pour simplifier et optimiser le développement logiciel. Par cette démarche, nous espérons contribuer à la réflexion sur l'évolution des paradigmes de programmation et sur la manière dont les langages de programmation peuvent évoluer pour mieux répondre aux défis contemporains du développement logiciel.

2 Étude des Design Pattern

2.1 Qu'est-ce qu'un Design Pattern et pourquoi l'utiliser ?

Un Design Pattern, également appelé modèle de conception, est un élément fondamental de la programmation orientée objet. Il consiste en une infrastructure logicielle composée d'un ensemble restreint de classes destinées à résoudre des problèmes techniques spécifiques.

2.2 Histoire

Les Design Patterns, ou patrons de conception, sont des solutions standardisées pour les problèmes courants de conception en programmation orientée objet. Le concept de Design Pattern trouve son origine dans le domaine de l'architecture, principalement inspiré par les travaux de l'architecte et théoricien du design Christopher Alexander dans les années 1970. Il vise à faciliter la création d'applications en appliquant des processus de conception similaires à ceux utilisés dans la conception de formes architecturales. Cette idée a été étendue à d'autres domaines tels que l'anthropologie, l'histoire de l'art, et bien entendu, l'informatique.

Mais c'est spécifiquement en 1987 que Kent Beck et Ward Cunningham ont réalisé que le concept de Design Pattern pouvait être appliqué à la programmation. Ils ont alors développé cette idée afin de l'adapter à la résolution de problèmes récurrents en informatique. Ces patterns sont devenus une partie fondamentale de la conception logicielle, offrant un langage commun et des solutions éprouvées pour des problèmes de conception courants.

Ils ont été popularisés par le livre *Design Patterns : Elements of Reusable Object-Oriented Software*, publié en 1994 dans le *Journal of Object-Oriented Programming* par le "Gang of Four", coécrit par quatre auteurs : Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides (qui met en avant un certain nombre de Design Patterns dans le développement logiciel que ses auteurs considéraient comme très fiables et efficaces). Ces patterns sont basés sur les principes de l'architecture logicielle et offrent un cadre pour résoudre des problèmes de conception de manière répétable et efficace.

L'utilisation des Design Patterns permet aux développeurs de communiquer efficacement, de réutiliser des solutions éprouvées et d'assurer une conception logicielle cohérente et maintenable. Ils sont devenus un élément essentiel de la boîte à outils de tout développeur logiciel.

2.3 Ses avantages

L'avantage de recourir à un Design Pattern est évident. Il accélère considérablement le processus de développement en fournissant des modèles de conception éprouvés. Cela permet de gagner du temps en évitant de réinventer la roue à chaque nouveau problème rencontré.

Étant conçu pour résoudre des problèmes courants, un Design Pattern permet de prédire et de traiter les difficultés potentielles dès les premières étapes du développement. De plus, la normalisation apportée par l'utilisation de Design Patterns améliore la lisibilité du code, facilitant ainsi sa maintenance et sa compréhension.

Grâce aux Design Patterns, les solutions sont documentées en s'appuyant sur les meilleures pratiques et les enseignements tirés des expériences antérieures. Les divers composants logiciels sont optimisés dans la mise en œuvre de ces modèles, ce qui accélère les processus faisant intervenir plusieurs éléments. Les développeurs peuvent ainsi utiliser efficacement le langage qu'ils maîtrisent pour appliquer ces solutions.

2.3.1 Partage du savoir

Dans le domaine de l'informatique, tant l'architecte informatique que le programmeur utilise le même langage lorsqu'il s'agit de Design Patterns. L'architecte n'a qu'à mentionner le nom du modèle pour

que le programmeur comprenne immédiatement de quoi il est question, évitant ainsi de longues explications. Il est donc crucial pour les professionnels de maîtriser les principaux types de Design Pattern. Les Design Patterns peuvent être intégrés à toutes les phases du processus de programmation. Ils servent de guide lors de l'écriture du code source, mais sont également fréquemment utilisés après coup, car leur efficacité permet de les utiliser comme modèles pour connecter des modules de code déjà existants.

Pendant la programmation, les Design Patterns sont particulièrement utiles pour identifier des similitudes avec des modèles déjà établis, permettant ainsi d'anticiper et d'éviter les erreurs potentielles. Ils sont généralement employés lorsque le développeur nécessite une certaine flexibilité dans son approche. Chaque Design Pattern adopte une approche unique. Les stratégies incluses dans chaque modèle sont donc distinctes et ne se ressemblent pas d'un Design Pattern à un autre. Cette diversité permet de résoudre chaque problème de manière organisée et efficace en appliquant le modèle approprié, déjà éprouvé. De plus, il est également possible de combiner différents modèles pour élaborer des solutions plus complexes.

2.3.2 Variétés de Design Pattern

Les Design Patterns se déclinent en plusieurs modèles, avec de nouveaux concepts qui émergent constamment pour répondre aux besoins évolutifs. Parmi eux, le modèle **Composite** est notable, utilisé pour structurer des hiérarchies arborescentes, telles que la représentation de dossiers contenant des sous-dossiers et des fichiers. Les éléments feuilles et composites suivent une même interface logicielle, simplifiant leur manipulation à chaque utilisation. Les Design Patterns sont divisés en trois catégories principales :

- **Créationnels** : concernent le processus de création d'objets, simplifiant la manière dont les objets sont créés, composés et représentés. Exemples : **Singleton**, **Factory**, **Builder**, **Prototype**.
- **Structuraux** : se rapportent à la composition des classes ou des objets, facilitant la conception en identifiant des manières simples de réaliser des relations entre les entités. Exemples : **Adapter**, **Decorator**, **Facade**, **Composite**.
- **Comportementaux** : se concentrent sur la communication entre les objets, améliorant la flexibilité et la répartition des responsabilités entre eux. Exemples : **Strategy**, **Observer**, **Command**, **Iterator**. Ce modèle élimine la nécessité de distinguer entre objets primitifs et conteneurs, offrant une gestion uniforme des objets individuels et de leurs agrégats.

2.4 Anti-pattern

Un anti-pattern en génie logiciel, gestion de projet et processus commerciaux est une réponse courante à un problème récurrent qui est généralement inefficace et risque d'être hautement contre-productive. Le terme, inventé en 1995 par le programmeur informatique Andrew Koenig, il a été inspiré par le livre *Design Patterns : Elements of Reusable Object-Oriented Software*. Un autre article en 1996 présenté par Michael Ackroyd lors de la conférence *Object World West* a également documenté des anti-patterns.

Cependant, c'est le livre *AntiPatterns : Refactoring Software, Architectures, and Projects in Crisis* publié en 1998. Ce livre, écrit par William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, et Thomas J. Mowbray qui a à la fois popularisé l'idée et étendu sa portée au-delà du domaine de la conception logicielle pour inclure l'architecture logicielle et la gestion de projet. D'autres auteurs l'ont étendu pour englober des anti-patterns environnementaux, organisationnels et culturels.

Définition : Selon les auteurs de "AntiPatterns", il y a deux éléments clés à un anti-pattern qui le distinguent d'une mauvaise habitude, d'une mauvaise pratique ou d'une mauvaise idée :

1. L'anti-pattern est un processus, une structure ou un modèle d'action couramment utilisé qui, malgré son apparence initiale d'être une réponse appropriée et efficace à un problème, a plus de mauvaises conséquences que de bonnes.

2. Une autre solution existe pour le problème auquel l'anti-pattern tente de répondre. Cette solution est documentée, répétable et prouvée, efficace là où l'anti-pattern ne l'est pas.

Un guide pour ce qui est communément utilisé est une "règle des trois" similaire à celle des patterns : pour être un anti-pattern, il doit avoir été observé se produire au moins trois fois. Les anti-patterns se manifestent souvent par une lenteur excessive du logiciel, des coûts de développement ou de maintenance élevés, des comportements anormaux et la présence de bugs. Parmi eux, les Grey-Patterns méritent une mention particulière, car leurs avantages ou inconvénients ne sont pas clairement établis.

Voici quelques exemples d'anti-patterns de développement :

1. Abstraction Inverse : Créer une interface logicielle qui offre des fonctionnalités complexes au lieu des fonctionnalités simples nécessaires, obligeant les utilisateurs à utiliser des fonctions complexes pour des tâches simples.
2. Action à Distance : Utilisation excessive de variables globales ou de fortes dépendances entre les objets.
3. Ancre de Bateau : Conserver des composants de code inutilisés pour des raisons politiques, dans l'espoir qu'ils seront utiles à l'avenir.
4. Attente Active : Utilisation d'une boucle d'attente qui vérifie périodiquement une condition jusqu'à ce qu'elle soit satisfaite, ce qui entraîne un gaspillage de ressources.
5. Interblocages et Famine : Problèmes de performance ou de plantages dus à une mauvaise gestion des ressources concurrentes.
6. Erreur de Copier-coller : Duplication de code entraînant des incohérences et une maintenance difficile.
7. Programmation Spaghetti : Code difficile à comprendre et à modifier en raison d'une structure complexe et entrelacée.
8. Réinventer la Roue (Carrée) : Création d'une solution complexe ou inefficace alors qu'une solution existante serait plus appropriée.
9. Surcharge des Interfaces : Multiplication des interfaces utilisateur offrant des fonctionnalités similaires.
10. L'Objet Divin : Classe logicielle contenant trop de responsabilités, ce qui nuit à la modularité.
11. Le Panier de Données : Création d'une structure de données complexe contenant des données inutilisées par les fonctions qui y accèdent, rendant la maintenance difficile.
12. Vous N'en Aurez Pas Besoin (YAGNI) : Implémentation prématurée de fonctionnalités non nécessaires.

Anti-patrons architecturaux :

1. ArchitectureAsRequirements : Cette erreur se produit lorsqu'une architecture est spécifiée uniquement par préférence ou parce qu'elle est nouvelle, alors qu'elle n'est pas nécessaire et que le client n'en a pas exprimé le besoin.
2. ArchitectureByImplication : Ce problème survient lorsque l'architecture utilisée par un projet n'est pas documentée ni spécifiée.
3. Coulée de lave : La coulée de lave se produit lorsqu'une partie de code encore immature est mise en production, entraînant des difficultés à la modifier par la suite.
4. Syndrome du Deuxième Système : Lorsqu'un système est réécrit avec trop de confiance, il peut aboutir à une architecture en sur-design qui devient lourde et inadaptée.
5. Marteau doré : Cette conception consiste à réutiliser une technologie familière de manière obsessionnelle, même si elle n'est pas adaptée à tous les problèmes.

Enfin, le pattern Singleton est également considéré comme un anti-pattern en génie logiciel pour certaines personnes en raison de ses implications négatives sur la modularité, la testabilité et l'évolutivité des applications. Le Singleton vise à garantir qu'une classe ne possède qu'une seule instance dans tout le système, ce qui peut sembler bénéfique pour le partage d'état global. Cependant, il introduit des dépendances cachées et rend les classes dépendantes fortement couplées au Singleton, rendant

ainsi le code difficile à maintenir et à tester. Les singletons ont tendance à créer un état global implicite, ce qui rend le comportement de l'application difficile à prévoir et à comprendre. De plus, lorsqu'il est mal implémenté, le Singleton peut entraîner des problèmes de concurrence dans les environnements multithread, introduisant des bugs difficiles à détecter. Dans l'ensemble, en raison de ces effets négatifs sur la qualité du code et la flexibilité du système, le pattern Singleton est généralement évité au profit d'approches telles que l'injection de dépendances pour gérer les instances uniques de manière plus contrôlée et modulaire.

Il est essentiel d'identifier et d'éviter ces anti-patterns lors de la conception et du développement logiciel afin de garantir la qualité et la maintenabilité du code.

2.5 Processus de création Design Patterns

La création de nouveaux Design Patterns (DP) est un processus dynamique et itératif qui implique souvent la collaboration de divers acteurs de la communauté du développement logiciel. Voici un aperçu des principales étapes et des acteurs impliqués dans ce processus :

1. **Identification des besoins :** Les développeurs, architectes logiciels et chercheurs observent les schémas récurrents dans les problèmes de conception logicielle. Ils identifient les lacunes dans les DP existants et les domaines sur lesquels de nouveaux DP pourraient être bénéfiques.
2. **Recherche et exploration :** Les personnes intéressées entreprennent des recherches approfondies pour explorer les solutions potentielles aux problèmes identifiés. Cela peut impliquer l'étude des travaux de recherche existants, l'analyse des meilleures pratiques de l'industrie et l'expérimentation avec différentes approches de conception.
3. **Prototypage et validation :** Les idées sont souvent mises en œuvre sous forme de prototypes ou de preuves de concept pour évaluer leur faisabilité et leur efficacité. Les prototypes sont testés dans des scénarios réels pour valider leur pertinence et leur utilité.
4. **Documentation et formalisation :** Une fois qu'une solution prometteuse est identifiée, elle est documentée de manière formelle sous la forme d'un nouveau DP. La documentation comprend généralement une description détaillée du problème résolu, de la solution proposée, des avantages et des inconvénients, ainsi que des exemples d'utilisation.
5. **Révision par les pairs :** La documentation du nouveau DP est souvent soumise à un processus de révision par les pairs au sein de la communauté du développement logiciel. Les commentaires et les suggestions des pairs aident à améliorer la qualité et la clarté du DP.
6. **Publication et diffusion :** Une fois finalisé, le nouveau DP est publié et diffusé au sein de la communauté du développement logiciel. Cela peut se faire via des articles de blog, des publications académiques, des présentations lors de conférences, des contributions à des bibliothèques de DP en ligne, etc.
7. **Adoption et évaluation :** Les développeurs commencent à utiliser le nouveau DP dans leurs projets et évaluent son efficacité dans la résolution des problèmes de conception spécifiques. Les retours d'expérience sont collectés pour informer les futures itérations du DP et son éventuelle modification ou amélioration.

Ce processus de création de nouveaux DP est itératif et continu, avec des opportunités constantes d'apprentissage et d'amélioration. L'implication active de la communauté du développement logiciel est essentielle pour garantir que les nouveaux DP répondent efficacement aux besoins changeants de l'industrie et de la technologie.

2.6 Futur des Design Pattern

Le futur des Design Patterns est étroitement lié aux tendances de l'industrie du logiciel et aux avancées technologiques. L'adaptabilité, la pertinence et la capacité d'innovation sont des éléments essentiels à la popularisation des nouveaux Design Patterns.
Voici quelques tendances et évolutions possibles que l'on peut envisager pour les Design Patterns dans les années à venir :

Principes SOLID et Design Spéculatif : Le design spéculatif et les principes SOLID ne sont pas des DP au sens traditionnel, mais ils sont des approches et des principes qui guident la conception de systèmes logiciels. Ils peuvent influencer ou donner naissance à de nouveaux DP à mesure que les technologies évoluent.

- **Single Responsibility Principle (SRP)** : Une classe ne devrait avoir qu'une seule raison de changer, ce qui signifie une seule responsabilité.
- **Open/Closed Principle** : Les entités logicielles (classes, modules, fonctions, etc.) doivent être ouvertes à l'extension, mais fermées à la modification.
- **Liskov Substitution Principle** : Les objets d'une superclasse doivent pouvoir être remplacés par des objets d'une sous-classe sans affecter la justesse du programme.

Design Patterns en IA : Les Design Patterns spécifiques à l'intelligence artificielle, notamment ceux en apprentissage machine, sont considérés comme de véritables Design Patterns dans le domaine de l'IA. Un exemple notable de la littérature sur ce sujet est le livre *Machine Learning Design Patterns* par Valliappa Lakshmanan, Sara Robinson et Michael Munn, qui présente une série de solutions éprouvées à des problèmes courants dans l'apprentissage machine. Ce livre couvre des patterns comme "Feature Store", "Repeatable Split", "Prediction Service", et d'autres qui sont spécifiques aux défis de l'apprentissage machine.

Design Patterns Émergents : Certains designs patterns émergents ne sont pas encore largement reconnus ou formalisés. Voici quelques concepts qui sont actuellement à l'étude et qui pourraient devenir des Design Patterns à l'avenir :

- **Immutable State Pattern** : Inspiré par les langages fonctionnels, ce pattern encourage l'utilisation d'états immuables pour éviter les effets de bord et faciliter le raisonnement sur le code.
- **Event Choreography** : Dans les systèmes distribués, ce pattern décrit comment les services peuvent coopérer sans dépendance directe, en utilisant des événements pour déclencher des actions entre eux.
- **Backpressure Pattern** : Utilisé dans les systèmes réactifs pour contrôler le flux de données et éviter la surcharge des consommateurs en régulant la production des producteurs.
- **Sidecar Pattern** : En architecture de microservices, ce pattern décrit comment un conteneur auxiliaire peut être déployé à côté d'un conteneur de service pour offrir des fonctionnalités comme la surveillance, la sécurité, ou la configuration.

Ces concepts sont explorés dans des projets avant-gardistes et des recherches en génie logiciel, et ils pourraient être adoptés plus largement à mesure que les pratiques et les technologies évoluent.

3 Étude de la programmation orientée objet

3.1 Fondements de la programmation orientée objet

La programmation orientée objet (POO) est un paradigme de programmation qui utilise des "objets" pour regrouper à la fois des données (appelées attributs ou propriétés) et des fonctions (appelées méthodes). Cette méthode facilite la modélisation d'entités réelles ou de concepts abstraits en objets logiciels, optimisant leur gestion de façon structurée et modulaire. Six principes fondamentaux soutiennent la POO :

- **Encapsulation** : ce principe consiste à confiner les données (variables) et les méthodes (fonctions) qui les manipulent dans une même unité, l'objet. L'encapsulation masque les détails d'implémentation à l'extérieur de l'objet, exposant uniquement les interfaces nécessaires pour interagir avec lui. Ce processus améliore la modularité, la sécurité et la gestion des accès aux données.
- **Héritage** : Le polymorphisme permet à une classe (ou objet) de reprendre les caractéristiques et les comportements d'une autre classe, dite de base ou parente. La classe héritière, appelée classe dérivée ou enfant, utilise cela pour augmenter la réutilisabilité du code, en créant de nouvelles classes à partir de classes existantes, tout en modifiant ou spécialisant le comportement hérité.
- **Polymorphisme** : Le polymorphisme autorise le traitement uniforme d'objets de différentes classes via une interface commune. Ainsi, un même nom de méthode peut invoquer différents comportements selon le type de l'objet concerné, augmentant la flexibilité et l'extensibilité du code.
- **Objet** : Un objet est une instance de classe qui combine des attributs et des méthodes pour manipuler ces attributs. En tant que fondements de la programmation orientée objet (POO), les objets modélisent des entités réelles ou conceptuelles.
- **Classe** : Une classe agit comme un modèle pour la création d'objets, définissant les attributs et méthodes communs à ces objets. Chaque objet créé selon ce modèle est une instance de la classe, encapsulant les données et méthodes spécifiques à un type d'objet.
- **Abstraction** : L'abstraction permet de se concentrer sur les caractéristiques essentielles d'un objet ou concept, en éliminant les détails superflus. En POO, cela se traduit par la définition de classes et interfaces qui capturent l'essence d'un concept, masquant les spécificités de mise en œuvre. Ce principe facilite la conception de systèmes plus modulaires, flexibles et compréhensibles.

3.2 Limites de la programmation orientée objet

Bien que la programmation orientée objet offre de nombreux avantages, elle présente également de nombreux défauts.

3.2.1 Rigidité et complexité

Une limitation majeure de la programmation orientée objet est qu'elle peut introduire de la rigidité et complexifier la conception logicielle. Comme illustré dans le schéma ci-dessous, on constate souvent une hiérarchie de classes étendue avec de multiples niveaux d'héritage. Cette architecture complexe peut rendre la compréhension du code difficile et nuire à sa modularité, puisque les classes sont fortement interdépendantes. Cette interdépendance excessive peut compliquer la maintenance et l'évolution du code, réduisant ainsi la flexibilité du système et sa capacité à s'adapter aux changements de besoins.

Dans cet exemple de diagramme de classes représentant différentes formes géométriques, on observe que la hiérarchie des classes est complexe, avec jusqu'à trois niveaux d'héritage pour certaines classes, comme la classe Rectangle. Cette structure en plusieurs couches peut entraîner des défis en termes de maintenance et d'évolutivité du code. En effet, chaque niveau d'héritage ajoute une couche de dépendance qui peut compliquer les modifications ultérieures et augmenter le risque d'erreurs lors de l'ajout de nouvelles fonctionnalités ou de la modification des comportements existants.

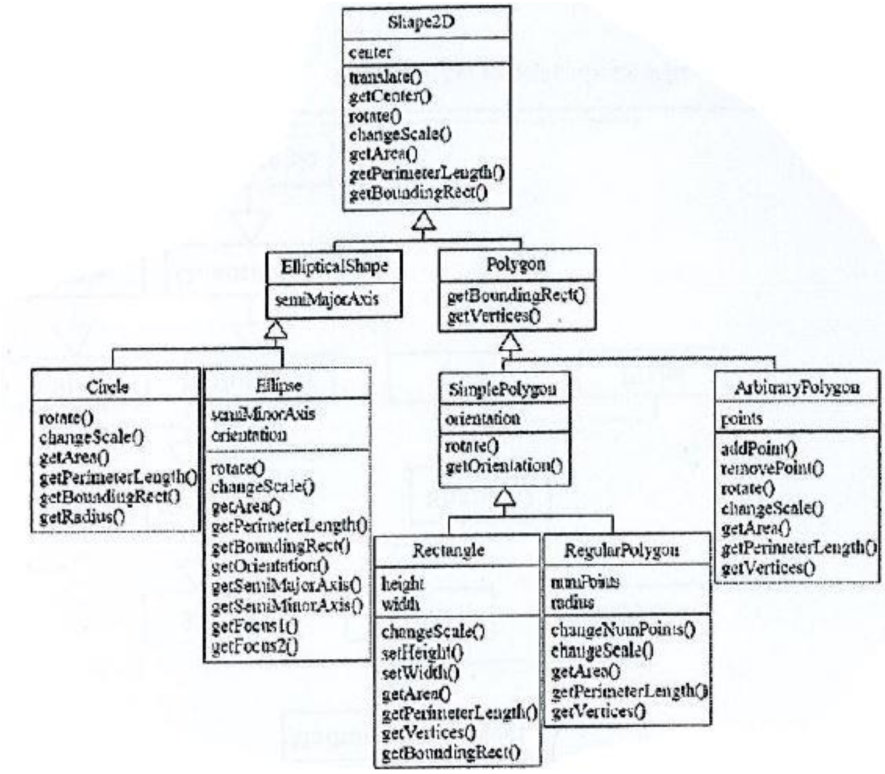

FIGURE 1 – Diagramme de classe de formes géométriques

3.2.2 Problème du diamant

Le problème du diamant est une complication spécifique à la programmation orientée objet, particulièrement présente dans les langages qui permettent l'héritage multiple, tels que C++ et certains aspects de Python. Ce problème survient lorsqu'une classe hérite simultanément de deux classes qui partagent une même classe de base. Cette structure forme schématiquement un diamant, d'où l'origine du terme. Voici une illustration de ce concept :

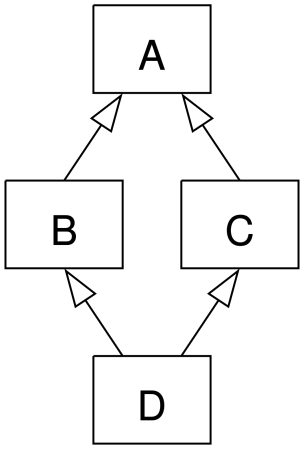

FIGURE 2 – Diagramme de classe en diamant

Dans ce scénario, les classes **B** et **C** héritent toutes deux de la classe **A**, et la classe **D** hérite à son tour de **B** et **C**. Un problème surgit lorsque la classe **A** possède une méthode que **B** et **C** ont modifiée indépendamment. Lorsque **D** tente d'appeler cette méthode, il n'est pas clair quelle version héritée devrait prévaloir, car **D** a des liens d'héritage avec **B** et **C**, toutes deux dérivées de **A**. Cette ambiguïté peut mener à des comportements imprévus ou à des conflits dans le code.

Pour contourner ce problème, les différents langages de programmation proposent diverses solutions. Python, par exemple, utilise un ordre d'héritage linéaire qui précise quelle méthode est utilisée en cas de conflit. En revanche, C++ permet une résolution plus explicite par le programmeur de la méthode à appeler. Java, quant à lui, évite ce dilemme en ne supportant pas l'héritage multiple, éliminant ainsi la source du problème.

3.3 Stratégies de conception en programmation orientée objet

Pour résoudre certains défis inhérents à la programmation orientée objet, différentes stratégies de conception ont été développées. Ces approches visent à améliorer la flexibilité, la maintenabilité et la scalabilité des applications. En explorant ces méthodes, les développeurs peuvent optimiser l'architecture logicielle et répondre plus efficacement aux besoins changeants des projets. En combinant judicieusement ces techniques, il est possible de surmonter les limitations traditionnelles de la POO et d'élargir les possibilités de conception logicielle.

3.3.1 Composition par héritage

La composition par héritage est une technique de programmation orientée objet où une classe utilise des fonctionnalités d'une autre classe en incluant des instances de cette classe dans sa propre structure. Contrairement à l'héritage classique où une classe hérite directement des propriétés et des méthodes d'une autre classe, la composition par héritage permet une relation plus flexible et modulaire entre les classes. L'idée est ici de favoriser la relation de composition par rapport à celle de l'héritage.

Dans le schéma d'héritage suivant, une classe abstraite nommée **Véhicule** sert de superclasse au sommet de la hiérarchie. Cette classe abstraite encapsule les attributs communs à différents types de véhicules. De cette superclasse **Véhicule**, deux sous-classes, **Voiture** et **Moto**, héritent des propriétés et méthodes communes. Par ailleurs, une classe **VoitureElectrique** est définie pour hériter spécifiquement de la classe **Voiture**, ajoutant ou modifiant des caractéristiques pour représenter des voitures électriques.

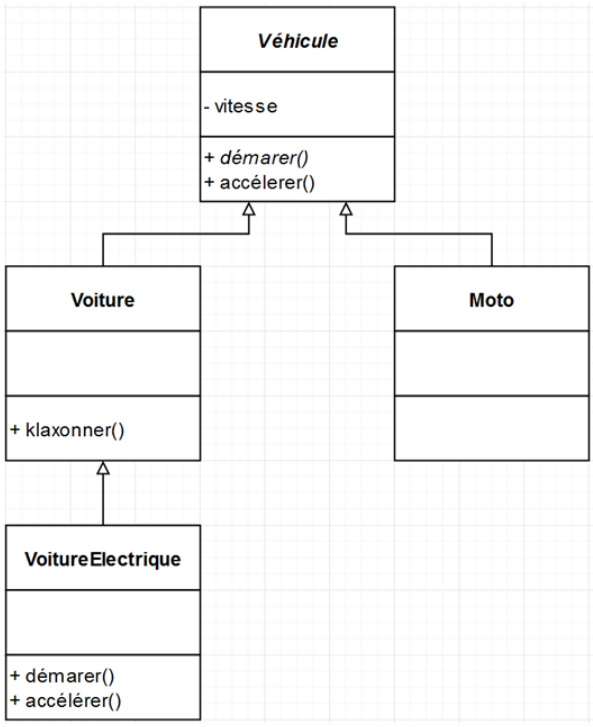

Figure 3 – Exemple de diagramme de classe avec héritage

Dans une approche utilisant la composition par héritage, on aurait à la place d'utiliser une classe abstraite, regroupé les attributs et logiques communes dans une classe dont **Voiture** et **Moto** auraient une composition. Quant à **VoitureElectrique**, elle aurait une composition vers **Voiture**. **Véhicule** aurait été à la place une abstraction, c'est-à-dire une interface dont toutes les classes auraient implémenté.

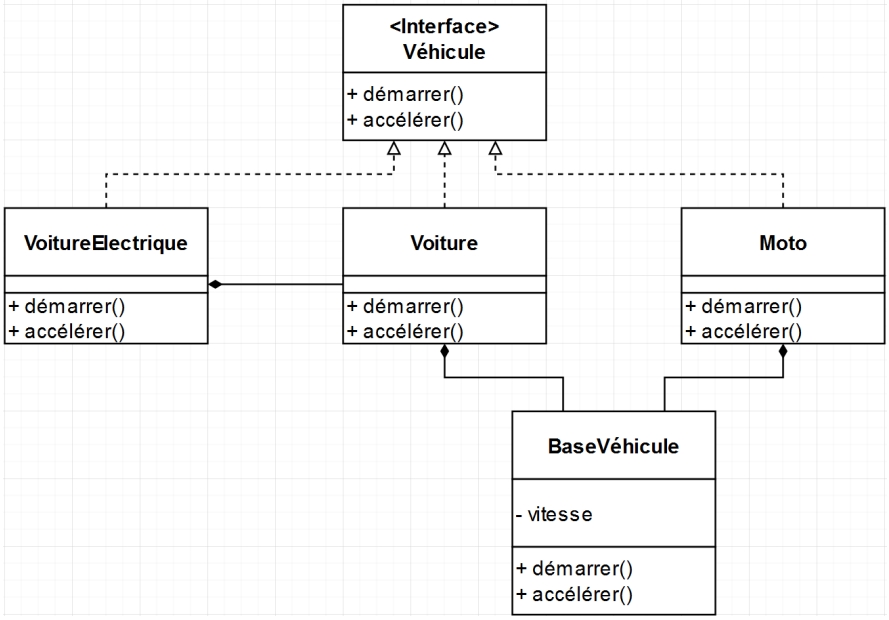

FIGURE 4 – Exemple de diagramme de classe avec composition par héritage

Dans une approche basée sur la composition plutôt que sur l'héritage, les attributs et fonctionnalités communs sont regroupés dans une classe à part, utilisée en composition par les classes **Voiture** et **Moto**. Ainsi, au lieu d'hériter d'une superclasse, chaque classe intègre une instance de cette classe commune pour accéder à ses propriétés et méthodes. De même, la classe **VoitureElectrique** serait composée à partir de la classe **Voiture**, reprenant et étendant ses fonctionnalités. Au lieu d'être d'une classe abstraite, **Véhicule** serait une interface définissant les méthodes que toutes les classes doivent implémenter, renforçant l'abstraction sans imposer une structure d'héritage.

Cette approche offre plusieurs avantages significatifs :

- **Flexibilité et modularité** : en utilisant la composition, les classes peuvent facilement intégrer ou modifier leurs comportements en incorporant différentes combinaisons de classes, ce qui favorise une conception plus flexible et modulaire.
- **Simplicité** : la structure hiérarchique est réduite à l'interface **Véhicule** et à ses différentes implémentations, sans héritages profonds. Cette simplification du modèle de classe aide à améliorer la compréhension, la maintenabilité et l'extensibilité du code.
- **Réutilisation du code** : bien que l'héritage regroupe le code commun en un seul endroit, la composition offre une alternative qui permet également de centraliser et de réutiliser le code sans les inconvénients souvent associés à l'héritage multiple ou profond.

Bien que l'exemple présenté soit simple, imaginons une situation impliquant un nombre bien plus grand d'entités. Dans ce cas, il devient nettement plus avantageux d'utiliser la composition plutôt que l'héritage afin de maintenir une hiérarchie de classes simple, au lieu de créer une structure hiérarchique, arborescente, complexe et profonde.

La composition par héritage offre également une solution au problème du diamant. Prenons l'exemple d'une classe abstraite nommée **Vehicule** dont deux classes, **Avion** et **Bateau**, y héritent. Ensuite, une classe **Hydravion** hérite à la fois de **Avion** et de **Bateau**. Le défi ici est l'ambiguïté qui survient lors de l'appel de la méthode **démarrer()** héritée des deux classes parentes, **Avion** et **Bateau**. Cette situation crée un conflit sur la méthode à exécuter, typique du problème de l'héritage multiple.

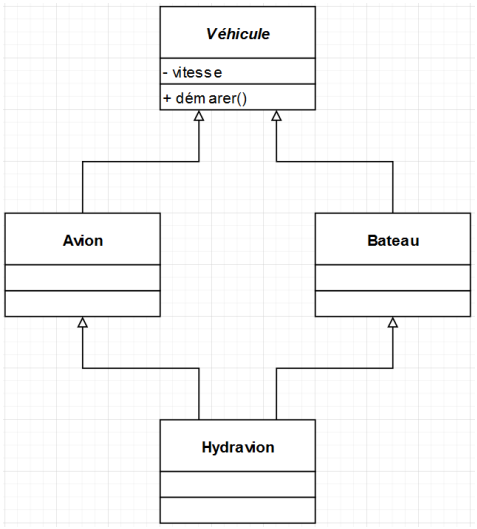

FIGURE 5 – Diagramme de classe avec problème du diamant

En optant pour la composition plutôt que pour l'héritage, le problème d'ambiguïté n'apparaît pas, car il n'y a plus de relation d'héritage directe. **Vehicule** est défini comme une interface plutôt qu'une classe abstraite. Les classes **Avion**, **Bateau** et **Hydravion** implémentent cette interface. Pour factoriser le code, **Avion** et **Bateau** utilisent une composition avec une classe auxiliaire qui contient les champs et implémentations nécessaires pour l'interface **Vehicule**. **Hydravion**, quant à lui, est composé d'**Avion** et de **Bateau**. Dans ce cadre, l'ambiguïté disparaît puisque **Hydravion** peut explicitement déterminer quelle implémentation de la méthode **démarrer()** appeler.

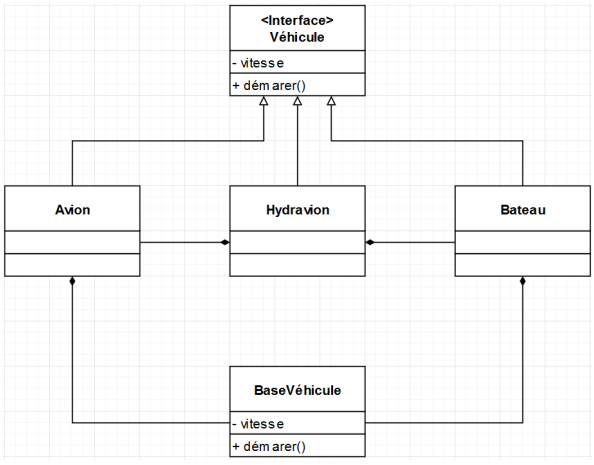

FIGURE 6 – Diagramme de classe avec héritage multiple via la composition par héritage

Cependant, la composition par héritage présente certains inconvénients par rapport à l'utilisation de l'héritage :

- **Suppression des hiérarchies intermédiaires** : par exemple, dans le cas de la composition plutôt que de l'héritage, un **Hydravion** n'est plus catégorisé comme un **Avion**, car il ne partage plus de relation d'héritage directe.
- **Plus de répétitions de code par rapport à l'héritage** : en effet, chaque classe qui utilise la composition doit gérer explicitement ses propres références aux composants qu'elle intègre plutôt que d'hériter directement de ces fonctionnalités. Cela peut entraîner une duplication de code pour l'initialisation et la gestion de ces composants dans chaque classe, alors que l'héritage aurait permis une centralisation plus efficace et moins redondante des comportements et attributs communs.

Effectivement, ce scénario souligne un dilemme classique en programmation orientée objet : le choix entre l'utilisation de l'héritage et celle de la composition.

- L'héritage permet souvent de réduire le volume de code, car les fonctionnalités sont implicitement héritées à travers la hiérarchie des classes. Cependant, cela peut nuire à la clarté du code et à la traçabilité des comportements, tout en complexifiant les relations hiérarchiques entre les classes.
- La composition améliore la clarté en isolant les fonctionnalités dans des composants distincts, bien que cela puisse entraîner une répétition accrue du code, puisque chaque classe doit explicitement gérer et intégrer ses propres composants. Cela simplifie également les relations hiérarchiques entre les classes, facilitant la maintenance et l'évolution du système.

Certains langages de programmation modernes, tels que Go ou Rust, ont choisi de ne pas inclure l'héritage dans leur conception. Malgré cela, ils sont toujours considérés comme des langages orientés objet, car ils intègrent d'autres principes fondamentaux de ce paradigme, tels que les classes, les objets, l'abstraction, le polymorphisme et l'encapsulation.

Cette décision soulève une question importante sur les pratiques en programmation orientée objet : l'héritage est-il à l'origine de nombreux problèmes associés à ce paradigme ?

3.3.2 Duck Typing

Le "Duck typing" est un concept utilisé en programmation, notamment dans les langages de programmation dynamiques comme Python. Ce principe repose sur l'idée que l'important n'est pas le type d'un objet en soi, mais les méthodes et propriétés qu'il possède.

L'expression vient de l'adage "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck" (Si ça ressemble à un canard, nage comme un canard et cancane comme un canard, alors cela doit probablement être un canard). En programmation, cela signifie que si un objet peut effectuer les actions requises (comme nager ou cancaner), alors il peut être traité comme tel, quelle que soit sa classe ou son type.

Dans le contexte de la programmation, cela permet une grande flexibilité et simplifie le code en réduisant le besoin de contrôles de type stricts. Par exemple, dans une fonction qui attend un objet qui peut "quacker", tout objet possédant une méthode quack peut être passé à cette fonction, sans se soucier de sa classe d'origine.

Le duck typing est particulièrement utile pour écrire du code qui est facile à comprendre et à maintenir, car il permet aux développeurs de se concentrer sur les capacités des objets plutôt que sur leur type exact. Cependant, cela peut aussi conduire à des erreurs moins prévisibles, qui ne seront détectées qu'à l'exécution si un objet ne répond pas aux attentes en termes de méthodes ou d'attributs nécessaires.

Pour illustrer ce concept avec un exemple pratique, considérons une interface nommée **Movable** et trois classes qui l'implémentent : **Animal**, **Véhicule**, et **Robot**. Dans la plupart des langages de programmation, par exemple, il est nécessaire de spécifier explicitement que ces trois classes implémentent l'interface **Movable** en utilisant le mot-clé *implements*. Cette approche définit clairement la hiérarchie et assure que toutes les classes respectent le contrat défini par l'interface.

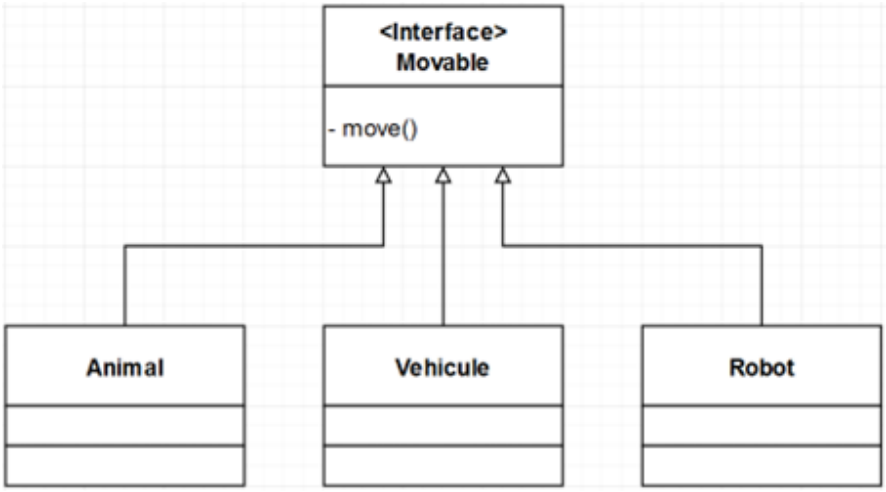

FIGURE 7 – Diagramme de classe avec implémentation

L'idée du Duck Typing est que la hiérarchie se construise de manière implicite, sans nécessiter de déclarations formelles des relations entre les classes. Ainsi, plutôt que de définir explicitement des interfaces ou des héritages, les objets sont évalués en fonction de leur capacité à répondre à certaines méthodes ou propriétés, ce qui simplifie l'intégration de divers types d'objets dans des fonctions ou des systèmes sans contraintes de type rigides.

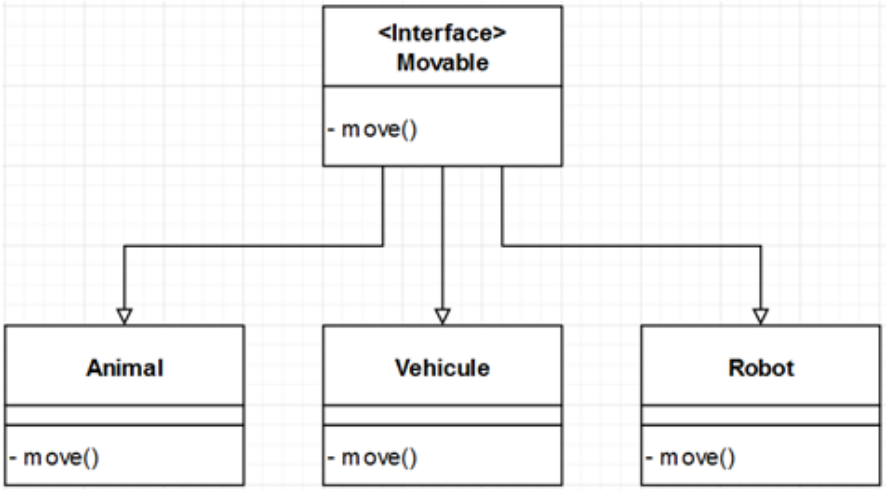

FIGURE 8 – Illustration du Duck Typing

Dans cet exemple, il n'est pas nécessaire pour les classes **Animal**, **Véhicule** et **Robot** de déclarer explicitement qu'elles implémentent l'interface **Movable**. Du moment qu'elles possèdent toutes les méthodes définies par l'interface, elles sont automatiquement considérées comme des implémentations de Movable.

Les langages de programmation où il est possible d'utiliser le Duck Typing sont :

- **Python** : En Python, il est possible d'invoquer n'importe quelle méthode ou attribut sur n'importe quel objet, à condition que cet objet possède cette méthode ou cet attribut au moment de l'exécution.
- **Ruby** : Ruby implémente également le Duck Typing, permettant une manipulation flexible des objets grâce à la vérification des méthodes et attributs à l'exécution plutôt qu'à la compilation.
- **Go** : Bien que statiquement typé, Go propose une forme de Duck Typing à travers ses interfaces. Une variable de type interface peut contenir n'importe quel type qui implémente les méthodes déclarées par cette interface, sans nécessiter une déclaration explicite de conformité de la part des types.

Un excellent exemple de Duck Typing dans de nombreux langages de programmation est l'utilisation des foncteurs. Un foncteur, dans ce contexte, se réfère à tout objet qui peut être utilisé ou appelé comme s'il s'agissait d'une fonction. Cela signifie que l'objet doit implémenter une méthode spécifique qui lui permet d'être invoqué. Cette capacité est un parfait exemple de Duck Typing, car elle ne dépend pas du type de l'objet, mais de la présence de cette méthode d'invocation.

Par exemple, en Python, les objets qui définissent la méthode __call__() peuvent être utilisés comme des fonctions. Cette caractéristique permet aux développeurs de créer des classes qui se comportent comme des fonctions, offrant ainsi une flexibilité accrue dans la manière dont les fonctions sont définies et manipulées.

De même, en C++, les classes peuvent surcharger l'opérateur () pour créer des instances qui peuvent être appelées comme des fonctions. Cela facilite l'écriture de code qui utilise des objets de manière interchangeable avec des fonctions normales, favorisant des patterns de conception comme les callbacks et les gestionnaires d'événements.

Cette approche de programmation souligne le principe du Duck Typing : si un objet se comporte comme une fonction, il peut être utilisé comme tel, indépendamment de son type sous-jacent. Cette flexibilité est particulièrement utile dans les langages de scripts et dynamiques, où il est souvent nécessaire de manipuler des objets de manière générique sans se préoccuper de leur classe spécifique.

4 Application pratique : AngryBalls avec modèle ECS

4.1 Projet AngryBalls

Durant le semestre précédent, dans le cadre du cours « Design Pattern », notre projet intitulé "AngryBalls" consistait à concevoir et à développer une application graphique. Cette application mettait en scène un plateau où des billes en 2D interagissaient entre elles à travers divers comportements tels que les collisions avec d'autres billes ou les murs et les frottements. Pour gérer l'ajout dynamique de ces comportements, nous avons utilisé le Design Pattern Décorateur.

L'emploi de ce Design Pattern a effectivement résolu plusieurs problématiques de modularité et d'extension des fonctionnalités sans altérer le code existant. Toutefois, cette approche présente des limites notables :

- **Retrait de comportements** : Enlever un comportement spécifique est complexe et minutieux, car cela implique la suppression d'un maillon précis dans une liste chaînée, ce qui peut introduire des erreurs ou des comportements inattendus.
- **Clonage d'une bille** : Le clonage requiert la reproduction de chaque maillon constituant la bille, rendant l'opération coûteuse en termes de performance et de gestion de la mémoire.
- **Structure lourde** : La présence d'une liste chaînée pour chaque comportement induit une surcharge en termes de performances, car chaque interaction entre les billes nécessite de traverser ces listes, augmentant ainsi le temps de traitement.

Ces limitations nous ont incités à explorer une alternative : le modèle Entité-Composant-Système (ECS), connu pour sa flexibilité et sa facilité à gérer des systèmes complexes en décomposant les fonctionnalités en composants réutilisables et en systèmes indépendants. Ce modèle promet une solution plus élégante et performante pour gérer les dynamiques complexes du jeu "AngryBalls".

4.2 Définition du modèle ECS

Le modèle Entité-Composant-Système (ECS) est une architecture logicielle utilisée principalement dans le développement de jeux vidéo, mais également dans d'autres domaines où la gestion efficace d'une grande quantité d'objets est nécessaire. Ce modèle se distingue par sa flexibilité et son efficacité, permettant de construire des systèmes complexes tout en maintenant une organisation claire et maintenable du code. Voici les trois principaux éléments de ce modèle :

1. **Entités** : Les entités sont des objets très légers, souvent représentés simplement par des identifiants uniques. Elles ne contiennent pas directement de données ou de logique, mais servent plutôt de conteneurs pour les composants. Dans un jeu, une entité pourrait être un personnage, un objet et dans le cas du projet "AngryBalls" une bille.
2. **Composants** : Les composants sont des conteneurs de données qui attachent des attributs spécifiques à une entité. Par exemple, un composant pourrait être la position, la vitesse, la santé, ou des caractéristiques graphiques d'une entité. Chaque composant est généralement structuré comme une structure de données simple sans méthodes propres, ce qui permet de les rendre extrêmement réutilisables.
3. **Systèmes** : Les systèmes sont où la logique du jeu ou de l'application est exécutée. Un système travaillera sur les entités qui ont un ensemble spécifique de composants. Par exemple, un système de mouvement pourrait uniquement affecter les entités qui ont à la fois des composants de position et de vitesse. Les systèmes permettent de traiter les données de manière efficace et isolée, en se concentrant uniquement sur les aspects du jeu qu'ils sont censés gérer.

Les avantages du modèle ECS par rapport au Décorateur sont :

- **Flexibilité** : Les entités peuvent être modifiées à la volée en ajoutant ou en retirant des composants sans perturber les systèmes existants.
- **Simplicité** : À l'inverse du Design Pattern Décorateur, où les comportements sont gérés par des chaînes d'objets décorateurs imbriqués, dans le modèle ECS, les comportements sont représentés par des classes de composants qui contiennent uniquement des données. La logique qui gère ces comportements est centralisée et uniformisée dans des systèmes dédiés. Cette séparation claire entre les données et la logique rend l'architecture non seulement plus simple, mais également plus facile à maintenir et à étendre.
- **Performance** : Les systèmes peuvent être optimisés pour traiter uniquement les entités qui les concernent souvent en utilisant des opérations sur des ensembles de données contiguës, ce qui est favorable à la performance sur les architectures modernes.

4.3 Refonte du projet

La mise en œuvre de notre projet d'Initiation à la Recherche consiste en la refonte du projet AngryBalls. L'objectif est de remplacer la conception originale basée sur le Design Pattern Décorateur par une architecture utilisant le modèle Entité-Composant-Système (ECS). Cette transition vise à exploiter les avantages du modèle ECS pour améliorer la modularité, la performance et la maintenabilité du projet.

Pour le langage de programmation, nous avons choisi d'utiliser Java pour plusieurs raisons :

- **Réutilisation des ressources existantes** : Les bibliothèques de calculs mathématiques fournies pour le projet AngryBalls peuvent être intégrées telles quelles, ce qui accélère considérablement le développement.
- **Démonstration de l'universalité du modèle ECS** : En utilisant Java, nous souhaitons démontrer que le modèle Entité-Composant-Système peut être efficacement appliqué dans différents langages de programmation, illustrant ainsi sa flexibilité et son adaptabilité.

5 Organisation

Résumé des Réunions

Date	Sujets Discutés	Décisions Prises
22/02/2024	Utilisation de Design Patterns dans le développement de plugins. (Money, COR, Observer...) Discussion sur l'observateur dans le contexte de la programmation Java.	Exploration des Design Patterns pour améliorer l'efficacité du code en Java. Test de développement d'un plugin pour formaliser et améliorer l'utilisation des Design Patterns.
8/04/2024	Exploration de frameworks backend, programmation orientée objet et représentation de graphes avec sommets et arêtes pour diverses applications. Problèmes liés au système consommateur-producteur en Java et utilisation des threads.	Choix de la mise en pratique d'un DP spécifique pour la mise en contexte.

6 Conclusion

Ce rapport a approfondi l'exploration des Design Patterns dans le contexte de la programmation orientée objet, soulignant leur importance cruciale pour améliorer la maintenabilité, la réutilisation du code et l'efficacité du processus de développement logiciel. À travers une variété d'exemples et d'études de cas, nous avons illustré comment les Design Patterns facilitent la résolution de problèmes complexes et renforcent la communication entre développeurs.

Nous avons aussi abordé les limitations inhérentes aux patterns traditionnels et discuté des défis que représentent les anti-patterns, susceptibles de compromettre l'efficacité du développement logiciel. Notre étude de cas, centrée sur le projet "AngryBalls", a démontré l'application concrète des Design Patterns et a révélé les nombreux avantages du modèle Entité-Composant-Système (ECS) comparativement à des approches plus conventionnelles telles que le Décorateur.

Quant à l'avenir des Design Patterns, il apparaît particulièrement prometteur, avec l'émergence de nouveaux paradigmes conçus pour répondre aux défis actuels, notamment dans le développement d'applications en intelligence artificielle et de systèmes distribués. Des principes tels que les principes SOLID et le design spéculatif continueront de guider les développeurs vers des solutions toujours plus robustes et évolutives.

En conclusion, à mesure que le domaine de la programmation orientée objet progresse, l'adoption et l'innovation en matière de Design Patterns resteront essentiels pour que les développeurs puissent répondre efficacement aux exigences évolutives du secteur logiciel. Il est crucial que la communauté des développeurs maintienne un engagement envers l'apprentissage continu et l'exploration de nouvelles approches de conception pour rester à la pointe de la technologie et assurer le succès de projets de plus en plus complexes.

7 Références

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley.
2. Beck, K., & Cunningham, W. (1987). Using Pattern Languages for Object-Oriented Programs. *Technical Report*, September 1987, OOPSLA'87 workshop on Specification and Design for Object-Oriented Programming.
3. Brown, W. J., Malveau, R. C., McCormick, H. W., & Mowbray, T. J. (1998). *AntiPatterns : Refactoring Software, Architectures, and Projects in Crisis*. Wiley.
4. Koenig, A. (1995). Patterns and Antipatterns. *Journal of Object-Oriented Programming*.
5. Lakshmanan, V., Robinson, S., & Munn, M. (2020). *Machine Learning Design Patterns*. O'Reilly Media.
6. Programmation orientée objet : *https ://en.wikipedia.org/wiki/Object-oriented_programming*
7. Problème du diamant : *https ://fr.wikipedia.org/wiki/Probl%C3%A8me_du_diamant*
8. Composition par héritage : *https ://en.wikipedia.org/wiki/Composition_over_inheritance*
9. Duck typing : *https ://fr.wikipedia.org/wiki/Duck_typing*
10. Modèle ECS : *https ://en.wikipedia.org/wiki/Entity_component_system*
11. Figure 1 : TD1 ACL - M1 Informatique - UFR MIM
12. Figure 2 : *https ://upload.wikimedia.org/wikipedia/commons/thumb/8/8e/Diamond_inheritance.svg/1024px-Diamond_inheritance.svg.png*
13. Logiciel utilisé pour les diagrammes de classes : *https ://app.diagrams.net/*