Université Paris 8
Master Arts
Mention : Arts Plastiques et Art Contemporain
Spécalité : Arts et Technologies de l'Image Virtuelle

Programmation : L'architecture dans l'application temps-réel

Christophe Delemis

Mémoire de Master 2
2013 - 2014

Université Paris 8
Christophe Delemis
2013 - 2014

Extrait

Ce mémoire traite de la mise en place d'outils de configuration de projets temps-réel en programmation à destination d'artistes ou de designers. Ces outils utilisent des fonctionnalités avancées permettant d'établir le fonctionnement d'une application, ainsi que son comportement lors de son exécution. Ils sont livrés avec des interfaces visuelles simples afin de rendre leur paramétrage ergonomique. Il sera question d'étudier comment ces fonctionnalités peuvent être implémentées de manière structurée dans un programme, et comment leur utilisation est possible à travers différents projets concrets.

Abstract

This master thesis deals with the setup of real-time projects configuration tools in programming for artists or designers.These tools use some advanced features to drive the application and its behaviour during its execution. They are provided with simple user interfaces to setting them ergonomically. We will explore how these features can be implemented in a structured way in a program, and how we can use them through different concrete projects.

Remerciements

A mon frère, Thomas, pour son apport de connaissances théoriques et techniques en programmation tous les jours.

A Arthur Juchereau, pour tous ses conseils sur la rédaction de ce mémoire, sa relecture intensive, et sa capacité à m'avoir supporté toute l'année lors des coups durs.

A Eddy Chopy, pour sa confiance et son soutien tout au long de ces trois années d'études supérieures.

A Cédric Plessiet, professeur à Arts et Technologies de l'Image, pour tous ses conseils en programmation et son soutien dans mon apprentissage.

A Marie-Hélène Tramus Chu-Yin Chen et Anne-Laure George-Molland, professeurs à Arts et Technologies de l'Image, pour leur écoute attentive de mon sujet de mémoire.

A Xavier Boissarie, dirigeant de la société Orbe à Paris, et Brice Herouart, chef de projet, pour leur accueil au sein de l'entreprise et de la confiance qu'ils m'ont apporté tout au long de l'année.

Et à tous mes amis de Limoges, qui prennent une place importante dans ma vie, et qui m'ont aidé à tenir le coup lors de l'écriture de ce mémoire.

Table des matières

1. INTRODUCTION

La programmation est omniprésente de nos jours. On la retrouve dans notre quotidien dans tous les systèmes informatiques et électroniques que nous possédons. Plusieurs domaines utilisent la programmation pour édifier des outils permettant à l'utilisateur de créer ses propres contenus, que ce soit dans la production graphique, sonore, ou bien le jeu vidéo.

Le domaine d'activité qui nous intéressera dans ce mémoire sera la programmation dans un projet temps-réel, et plus précisément les fondations des outils disponibles pour les artistes et designers dans le paramétrage de leurs applications. Nous traiterons alors ces outils d'un point de vue logiciel, complétés par des interfaces utilisateurs au service des designers.

L'application ou le jeu peut être amené à évoluer, ainsi que les outils demandés par les designers, et il devient nécessaire de réfléchir à un développement flexible et extensible, pouvant résister à de nombreuses modifications dans le temps. Pour cela nous mettons en place des structures de code, basés à la fois sur des théories et des techniques évolutives, le tout formant une architecture complète.

Nous étudierons dans un premier temps les bases de la programmation d'aujourd'hui, comment elle a été mise en place, et quels sont les outils mis à notre disposition pour développer des systèmes. Nous aborderons ensuite des cas concrets au moyens de deux projets distincts. Ces projets utiliseront comme base le moteur de jeu Unity3D, qui nous permettra d'élaborer des outils de développement spécifiques ainsi que leurs interfaces dédiées.

Le premier projet sera ma recherche personnelle tout au long de l'année pour étendre les fonctionnalités du moteur, tandis que le second projet traitera d'une expérience professionnelle au sein d'une entreprise, où nous aborderons la modification d'un code déjà existant afin de le rendre paramétrable et accessible pour un designer.

Ce mémoire s'adresse surtout aux jeunes développeurs désireux d'approfondir leur réflexion en programmation et en structure de code, en leur proposant ma vision et mon approche sur des besoins spécifiques.

2. L'ARCHITECTURE DANS LE DÉVELOPPEMENT SOFTWARE

Le développement software (ou développement logiciel) connaît sa première apparition dans les années 1940, mais il faudra attendre la fin des années 1960 pour que le terme devienne une profession à part entière, acceptée des programmeurs dans le monde.

Durant cette époque, de nombreux langages de programmation sont développés comme le FORTRAN (IBM, 1955) ou le COBOL (CODASYL, Conference on Data Systems Languages, 1959), afin de permettre aux programmeurs de «parler» avec l'ordinateur.

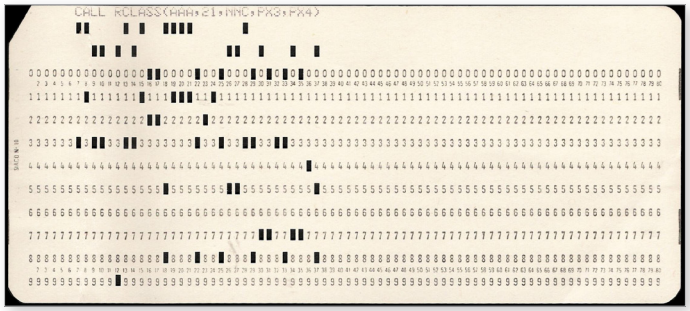

Fig.1. Un programme sauvegardé sur une carte perforée dans les années 1960

Il était naturel pour les développeurs que le langage utilisé correspondait à la manière dont le processeur traitait les informations, c'est à dire séquentiellement, suivant une suite d'opérations logiques bien définies. On parle alors de *programmation procédurale.*

En parallèle, le développement hardware (composants physique de l'ordinateur) connaît une expansion que l'on défini par la Loi de Moore, exprimée en 1965, qui dis que la puissance des ordinateurs doublerait tous les 18 mois.

Cette croissance permit au développement software de modifier son approche en utilisant toute la puissance disponible de l'ordinateur. Ainsi les programmes ne sont plus écrits sur des cartes perforées mais stockées directement dans l'ordinateur.

Dans les années 1970 apparaît la notion de *programmation structurée*, soutenue par Nicklaus Wirth (inventeur du langage Pascal), et de Edsger Dijkstra (connu notamment pour ses algorithmes sur les graphes). Cette notion recommande des organisations hiérarchique du code que l'on retrouve encore de nos jours par l'utilisation de mot clés tel que «while», «if... else...», tout en supprimant des instructions tel que le GOTO[1], énormément utilisée depuis la naissance du développement software.

1 GOTO est une instruction présente dans de nombreux langages de programmation avant la programmation structurée qui avait la particularité de réaliser des sauts d'exécution dans un programme vers une autre instruction, rendant la lisibilité du code très difficile.

La programmation structurée connaît un immense succès et sera acceptée par une majorité, donnant naissance à de nouveaux langages de programmations, notamment le C, très populaire encore aujourd’hui.

Malgré ces avancées, le monde est en crise jusqu’à la fin des années 1980. On l’appelle la *Software Crisis* (Crise du logiciel). Les projets sont jugés de mauvaise qualité, due souvent à un manque de budget ou de temps. Ils deviennent difficile à maintenir et il est temps pour les développeurs de trouver des solutions à ces problèmes.

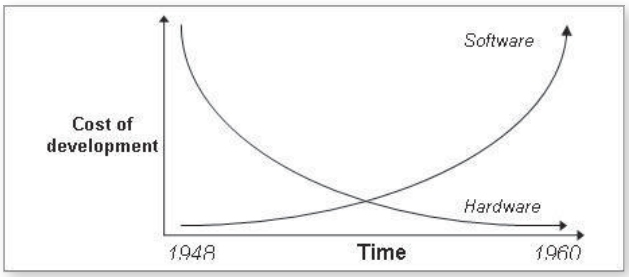

Fig.2. Coût du développement software et hardware au fil des années.

On parle alors de la *silver bullet* (balle d’argent), une technologie qui réglerai tous les problèmes évoqués. Malheureusement, il est décrété quelques années plus tard qu’il ne peut exister de silver bullet malgré de nombreuses tentatives.

Une des solutions était de transposer l’univers réel dans une solution informatique, en s’exportant du traitement binaire et logique du processeur, l’unité centrale de l’ordinateur opérant les calculs nécessaires à l’exécution du programme. Toutes les entités autour de nous pouvait être défini dans un ensemble de données, en leur donnant des caractéristiques propres ainsi que des actions pour les modifier.

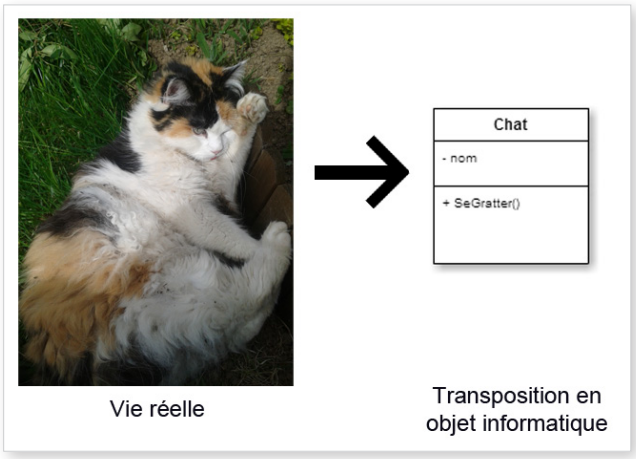

Fig.3. Représentation d’une entité réelle vers une entité informatique

Ce concept est connu sous le nom de *programmation orientée objet*, et sera au centre de ce mémoire.

La relation entre ces entités formait un système complexe, si bien qu’il était nécessaire d’ordonner le tout pour obtenir un programme fiable, maintenable, sécurisé, et facilement extensible.

C’est alors qu’apparaît les problématiques d’architecture dans le développement software. Comment structurer l’ensemble du système ? Comment les éléments doivent réagir entre eux ? Comment proposer une solution capable de résister à de nombreux changements?

Ces questions sont toujours d’actualité et de nombreuses solutions sont proposées chaque année. L’évolution de l’architecture dans sa globalité réside dans la recherche de nouvelles analyses (par des systèmes de modélisation) ainsi que de nouvelles techniques (par des concepts avancés du code).

Certaines architectures ont été pensées et conçues pour des domaines spécifiques, tel que le web, les bases de données, la gestion d’utilisateurs, etc. Il n’y a donc pas «une» architecture meilleure que les autres, mais des solutions basées sur l’expérience vis à vis d’un problème spécifique.

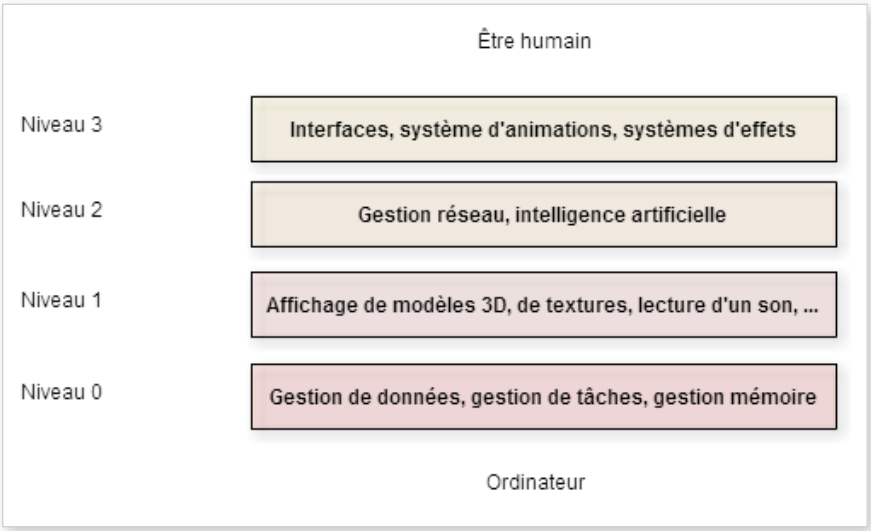

Fig.4. Exemple de niveaux de couches dans le domaine du jeu vidéo

Dans le monde du jeu vidéo, on peut noter différents niveaux (ou couches) de problématiques pour un produit fini. Plus le niveau est «bas», plus la structure est proche de la machine, et moins l’ergonomie humaine est prise en compte. A l’inverse, un niveau «haut» traitera les opérations d’un point de vue humain. Ouvrir une porte est compréhensible par l’homme, car il correspond à notre mode de vie et à notre perception de la vie, alors que procéder à une conversion d’un espace 3D vers un espace 2D via l’utilisation de matrices est plus compliqué à manipuler.

Afin de comprendre comment une architecture peut être mise en place, nous allons faire un bref récapitulatif de la programmation, du moins les points qui nous intéresserons pour la suite de ce mémoire. Nous verrons ensuite des concepts plus avancés et enfin comment ceux-ci s’organisent pour former un système complexe.

2.1. Programmation Orientée Objet

2.1.1 Avant la programmation orientée objet

La *programmation procédurale* est considérée comme un concept destiné à parler principalement à la machine. Le processeur traite ligne par ligne les séquences qui lui sont données et les exécute. Il est facile pour la machine de gérer ces informations, mais contraignante pour l'humain, qui l'oblige à anticiper le fonctionnement de l'ordinateur.

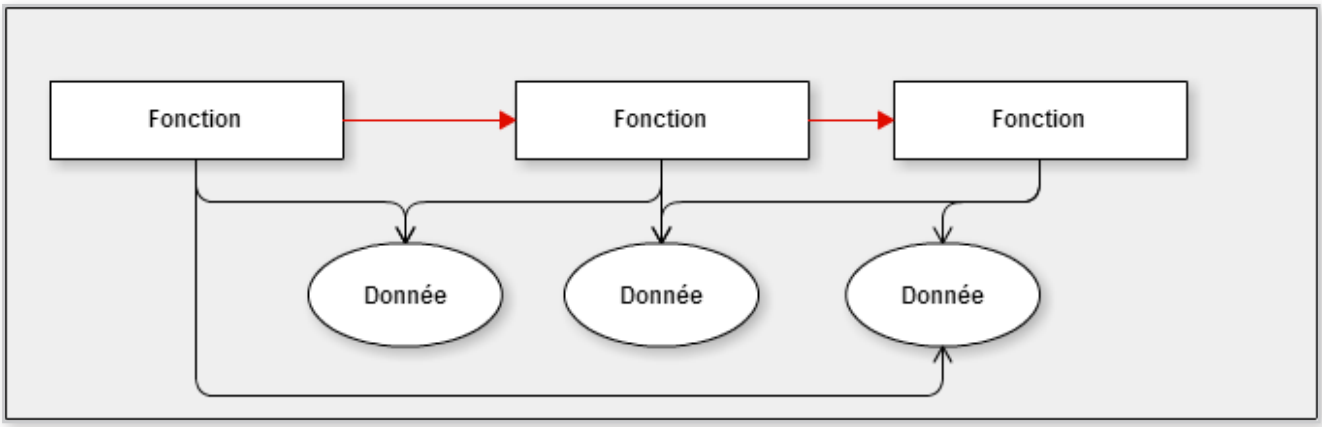

Fig.5. Représentation relationnelle en programmation procédurale

La *programmation procédurale* décompose le programme en fonctions ou *procédures* qui déterminent le cheminement logique des opérations à effectuer. Ainsi, à un instant «t», le programme est tenu de suivre une séquence de traitements. Des modifications d'états ou des calculs se réalisent lors de ces traitements. Parfois, ces procédures doivent aller récupérer des données dans une partie extérieure qui est partagée à l'ensemble du programme. On appelle ces données des «données globales», ou *variables globales*.

Si l'on considère que ces variables sont utilisés des centaines de fois dans tout le code, il devient alors très difficile d'opérer des modifications sur celles-ci, comme changer de nom la variable, car il faudrait alors remplacer le nom de cette variable dans toutes les fonctions qui l'utilisent. Ce principe s'appelle la *dépendance* et est une des contraintes les plus difficiles de gérer en programmation.

Considérons maintenant que plusieurs fonctions modifient la valeur d'une variable globale, comment savoir très exactement, lors de la phase de «debug»[1], qui a modifié cette valeur à l'instant précis et pourquoi ?

Il est fortement déconseillé d'avoir recours à des variables globales pour étendre la maintenabilité d'un programme. Il se peut cependant que dans certains cas l'on soit obligé d'y avoir recours.

1 Le debug (ou débogue en français) consiste à la maintenance d'un système ou d'une entité pour limiter les comportements involontaire du programme, comme un plantage.

2.1.2 Bref historique

On doit le concept d'objet en programmation au langage Simula 67, développé dans le début des années 1960 par Kristen Nygaard et Ole-Johan Dahl. C'était un langage destiné à produire des simulations d'explosions. Les développeurs se sont rendus compte que chaque explosions avait des caractéristiques différentes et qu'ils pouvaient les regrouper dans des entités similaires. Il apparaît alors la notion de classe, qui définit une catégorie ou un type, et d'instance de classe, qui définit un objet concret de cette catégorie.

Cette nouvelle approche a permis à Alan Kay de développer son langage SmallTalk à Xerox PARC en 1972, qui deviendra un langage d'une grande influence pour de nombreux langages futurs (tel que Objective-C, ou bien Java). Smalltalk introduit de nouveaux concepts dans la méthodologie, en plus des principes de base déjà fournit avec Simula 67, qui en fera un langage puissant.

Dans le milieu des années 1980 apparaît le C++, créé par Bjarne Stroustrup. C++ est un langage basé sur le C en ajoutant la dimension orientée objet. Il sera très vite accepté de la majorité des programmeurs grâce à sa capacité de traiter les problèmes de manière humaine tout en restant «proche de la machine». Ce langage est encore l'un des plus utilisés de nos jours, considéré comme un langage maître en terme de performance dans le développement software, bien qu'il nécessite une plus grande expérience que certains langages plus récents.

En 2002 sort le framework .Net (prononcé «dot net») par Microsoft, une librairie permettant de développer des applications Windows ou Web. Ce framework est aujourd'hui à sa version 4.5 et est directement intégré au langage de programmation C# (aussi développé par Microsoft).

> Comme nous allons utiliser le moteur de jeu Unity3D par la suite, nous allons nous focaliser sur le langage C# tout au long du mémoire, puisqu'il est directement supporté par le moteur.

Il est intéressant de lire le livre «Framework Design Guidelines» de Krzysztof Cwalina et Brad Abrams, tout deux membres fondateurs du framework .Net, et de découvrir les directions prises pour élaborer la conception de la bibliothèque.

2.1.3 Quelques principes de l’orienté objet

A l’inverse de la programmation procédurale, la *programmation orienté objet (POO)* décompose le programme en *objet*, principalement issue de la vie réelle, ayant des paramètres, ou *attributs*, et des actions disponibles permettant la modification de son état, sous formes de fonctions, appelées *méthodes*. On obtient alors un concept proche du raisonnement humain.

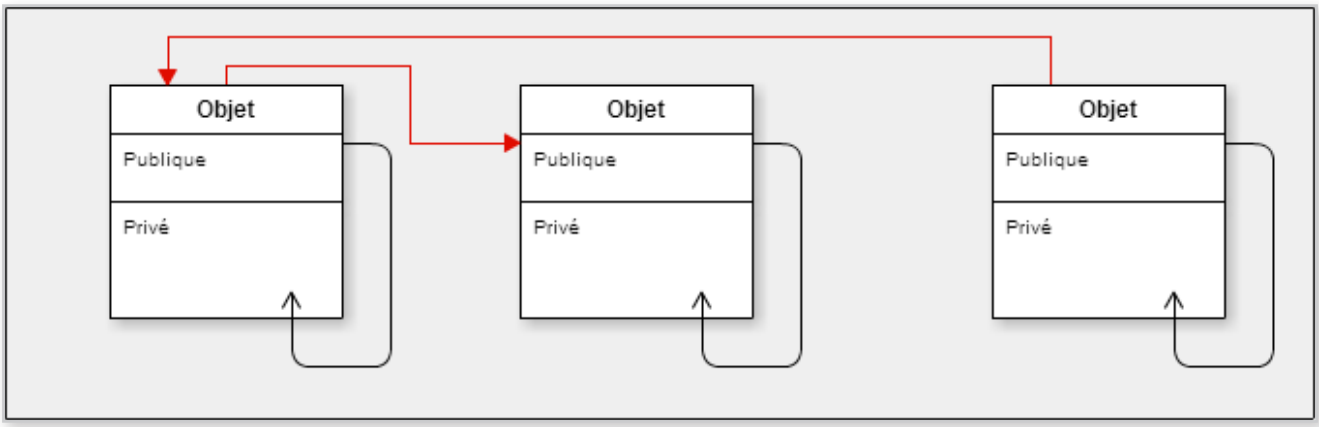

Fig.6. Représentation relationnelle en programmation orientée objet

Un objet est issue d’une *classe*, qui définit son patron de conception et sera considérée comme le «type» de l’objet. L’objet devient alors une instance de cette classe. Plusieurs objets peuvent être crée d’une même classe, si bien qu’ils deviennent unique dans le programme tout en partageant des caractéristiques. Une usine crée des voitures à la chaîne répondant à une conception unique pour un même modèle. Le modèle de la voiture correspond à la classe, tandis que la voiture physique créée est une instance, un objet. Un objet possède des actions à effectuer, appelées *méthodes*. Pour une voiture, ses méthodes pourraient être «Démarrer», «Arrêter», «Tourner», et ainsi de suite.

Un objet peut aussi contenir d’autres objets, créant ainsi une hiérarchie structurée. Une voiture peut être un objet, ayant elle même des objets de type «roues», «volant», etc.

Cette notion de hiérarchie amène à un principe élémentaire de la programmation orientée objet, l’*encapsulation*. L’encapsulation permet de définir quels éléments de l’objet sont disponibles publiquement à l’extérieur de celui-ci. Elle permet de différencier les actions disponibles pour les autres objets (comme démarrer la voiture) et de cacher son fonctionnement interne (allumer le moteur). On part du principe que l’objet demandant une action à un autre objet n’a besoin de que du résultat de l’opération.

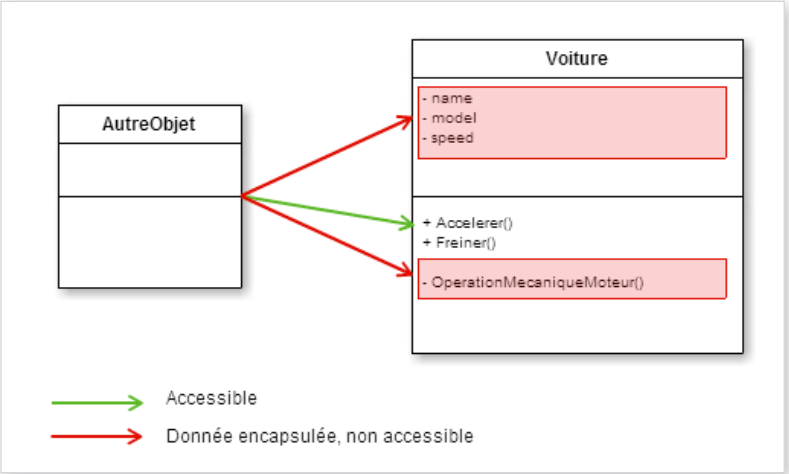

Fig.7. Principe de l’encapsulation en orienté objet

Dans l’exemple de la voiture, lorsque le conducteur appuie sur la pédale d’accélération, il n’a pas besoin de savoir toutes les opérations mécaniques, et encore moins besoin de les faire lui même, pour faire avancer son véhicule. Tout l’aspect mécanique a été encapsulé par l’objet «voiture».

Beaucoup de concepts avancés de la programmation orientée objet proviennent de l’encapsulation car elle permet de diminuer les dépendances dans l’application. En laissant privées toutes les parties non importantes pour l’utilisateur, il y a moins de risque d’une utilisation abusive de ces informations.

L’orienté objet apporte aussi la notion d’*héritage* et de *polymorphisme*, qui permet à des objets de garder des caractéristiques d’un type d’objet tout en ajoutant des nouveaux qui lui seront propre. Cela permet d’étendre les fonctionnalités de certains types d’objet.

Une «voiture» définie l’entité du véhicule, mais il existe différents types de voiture. Nous pourrions avoir des «voitures décapotables» ou bien des «4x4», qui possèdent tout autant les fonctions d’avancer ou de reculer, héritées du type «voiture».

Plus l’on développe des classes «filles» par héritage, plus le type d’objet devient spécifique pour l’utilisateur.

Le schéma suivant est basé sur une figure du livre de Hugues Bersini «La programmation orientée objet» et décrit comment une hiérarchie basée sur l’héritage pourrait être vu à différents niveaux. (page 225)

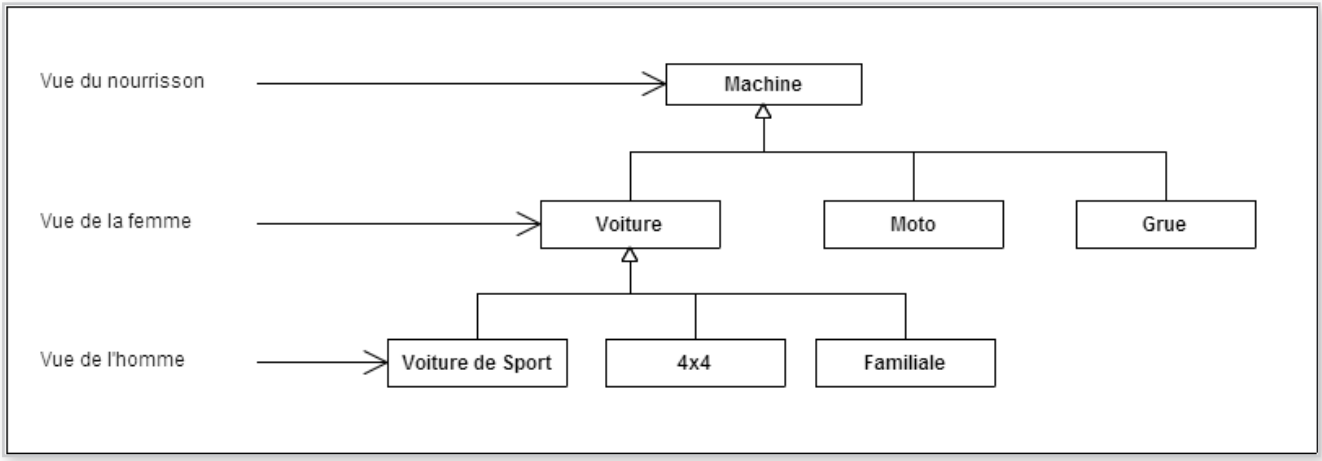

Fig.8. Exemple de hiérarchie à base d’héritage en orienté objet

On parle alors d’*abstraction*. L’abstraction existe lorsque l’on ne fait pas référence directement à un type précis.

Ici, le nourrisson voit la voiture comme une machine, et considère qu’elle fonctionne comme toute machine. La femme (n’y voyez aucun sexisme), décrit son véhicule comme étant une simple voiture, et connaît son fonctionnement de base pour circuler avec. Quand à l’homme, il a pleinement connaissance que sa voiture est une voiture de sport qui adhère bien au sol et peut se permettre de faire des dérapages avec.

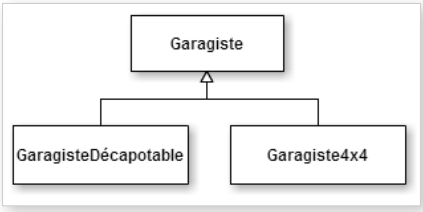

Fig.9. Exemple de hiérarchie pour un garagiste

En parallèle, un garagiste a besoin de recevoir une voiture pour travailler. Cette voiture peut être décapotable ou non, son travail consistera à la réparer comme n’importe quelle voiture. Il se peut qu’un problème très spécifique au modèle survienne, alors le garagiste devra être capable de réparer ce genre de modèle (comprenez qu’il devra avoir la capacité de recevoir un objet de type «voiture décapotable» par exemple), ce qui lui donnera une compétence spécifique. Mais un garagiste étant spécialisé dans les «4x4» ne saura pas traiter les décapotables. Les garagistes ayant les compétences de traiter ces genres de modèles seront avant tout des «garagistes».

On a alors une nouvelle hiérarchie d’héritage et d’abstraction, où les deux garagistes hériteront du métier de garagiste généraliste. Dans certains langages de programmation (notamment le C++), on a la possibilité de faire hériter une classe de plusieurs classes. On obtient alors du *multi-héritage*.

2.1.4 Analyse d’un système

Pour représenter un objet ou un ensemble d’objets liés, on utilise un système de modélisation appelée UML (Unified Modeling Language). UML propose des normes de modélisation d’un système objet à plusieurs niveaux, sous la forme de diagrammes :

- ◊ les diagrammes de classes,
- ◊ les diagrammes de séquences,
- ◊ les diagrammes d’activités,
- ◊ ainsi que les diagrammes de composants pour les plus courants.

Il est possible grâce à UML de discuter du système avec n’importe quelle personne d’une équipe beaucoup plus facilement, et de se représenter visuellement l’interaction des objets entre eux.

Le schéma suivant montre un diagramme de classe très sommaire représentant les relations entre des professeurs et étudiants au sein d’une même formation. L’étudiant rédige son mémoire, et les professeurs le corrige (le schéma ne montre que l’essentiel).

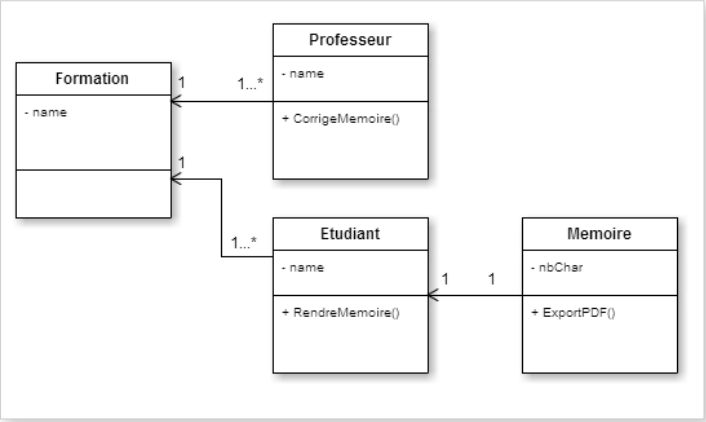

Fig.10. Exemple de diagramme de classe en UML

2.2. Structure de code

La programmation orientée objet apporte des concepts techniques dont le programmeur est libre d'utiliser pour créer ses classes et les relations entre elles, tel que l'encapsulation, l'abstraction, l'héritage, etc. Mises ensemble dans le programme, cela entraîne des compositions relationnelles et on parle alors de design de code, ou bien encore de patterns (patrons de conception). Cela pourrait s'apparenter à un puzzle où chaque pièce s'imbrique dans d'autres pour former un tableau complet et homogène.

Le design de code est formé à partir d'un besoin spécifique, il est généralement considéré comme une solution à un problème rencontré. Lorsqu'il n'est pas respecté, la structure entière devient instable et le programme peut être menacé. Lorsque toute la structure devient incontrôlable, nous obtenons ce qu'on appelle du «code spaghetti», en référence à l'aspect que peut avoir un plat de spaghettis dans une assiette, et chaque modification dans le programme entraîne de lourde répercussions dans l'ensemble du projet.

C'est alors que différents concepts avancés apparaissent pour garantir une plus grande souplesse dans la structure d'un programme. On parle souvent de design patterns, connu pour les vingt-trois patterns développés par ceux qu'on appelle le «Gang of Four» (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides), réuni dans un seul livre «Design Pattern : Elements of Reusable Object-Oriented Software» écrit dans les années 1980, bien que d'autres ont été développés plus tard par d'autres auteurs.

En parallèle, une autre forme de concept apparaît, sous le nom de «Principles». Ces principes sont une sorte de guide à suivre permettant une meilleure efficacité lors du développement d'un programme. Les plus connus sont SOLID et GRASP.

2.2.1 «Code spaghetti»

La notion de «code spaghetti» apparaît lorsque le programme entier repose sur des fondations très peu définies et que chaque nouvel élément essaie d'avoir sa place dans le programme bien que la structure de base n'y soit pas préparée. Il faut alors modifier petit à petit l'ensemble pour accepter ce nouvel élément.

Dans l'exemple du puzzle, on obtiendrait un tableau dont on accepterait que les pièces soient interchangeables pour correspondre à la forme désirée, sans se préoccuper du résultat final. Le tableau aurait donc une forme géométrique susceptible de fonctionner mais aucun rendu graphique.

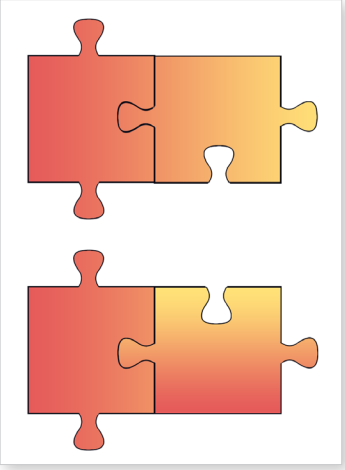

Fig.11. Représentation imagée du «code spaghetti»

Toutes ces erreurs de conception logiciel peuvent être renommés des «Anti-Patterns» et sont considérés comme contre productif. Ils bloquent généralement le bon fonctionnement de l'application. On note souvent des lenteurs dans l'exécution, ou bien la maintenance du programme se retrouve plus délicate qu'elle n'aurait dû. A mesure que le projet évolue, certaines parties du programme deviennent difficile à contrôler.

Il faut donc à tout prix éviter ce phénomène et réfléchir à des solutions adaptées, en réunissant les problématiques, et en organisant les concepts techniques de l'orienté objet à notre disposition pour former des designs solides. C'est le but principal des Design Patterns.

2.2.2 Design Patterns

Les design patterns, ou patrons de conception, déterminent la structure relationnelle d'un sous-système afin de répondre à une problématique ciblée. Ils organisent d'un point de vue technique la composition entre objets, la façon dont ils communiquent ainsi que leurs responsabilités dans le système.

Les premiers patterns à voir le jour ont été les vingt-quatre design patterns du «Gang of Four», composé de Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. Leur expérience leur a permis de concevoir des schémas de développement structurés, indépendamment du langage utilisé, et de les proposer aux programmeurs du monde entier pour répondre à des problématiques récurrentes.

Dans un système complexe, un design pattern permet généralement d'accroître le temps de développement et d'augmenter la qualité du résultat attendu. Leur principal objectif est de rendre l'ensemble flexible, maintenable et réutilisable.

Ils se découpent en trois grandes catégories : Creational, Structural et Behaviour patterns.

- ◊ Les **Creational patterns** (patrons de conception) définissent comment les objets sont instanciés (c'est à dire créés) et configurés dans le code.

- ◊ Les **Structutal patterns** (patrons de structure) définissent comment les objets sont organisés entre eux dans un projet à large échelle en se basant uniquement sur leur interface.

- ◊ Les **Behaviour patterns** (patrons de comportement) définissent comment les objets communiquent entre eux et se partagent les responsabilités tout en établissant des schémas type pour l'implémentation d'algorithmes.

Ce mémoire n'a pas pour but de décrire tous les design patterns existants, néanmoins vous pouvez en retrouver quelques uns utiles pour la suite en Annexe 2. Pour plus d'informations, les design patterns sont généralement bien documentés sur Internet où dans des livres dédiés.

Il est primordial de comprendre l’utilité de chaque design pattern, de la motivation qui a poussé à sa création, et de la solution apportée. Cependant, il est déconseillé d’utiliser les design patterns lorsque ce n’est pas nécessaire. Leur structure peut parfois être délicate à comprendre, et les programmeurs n’ayant pas connaissance du design utilisé pourrait avoir des difficulté à lire le code.

Ci-dessous un schéma UML représentant la structure du design pattern State.

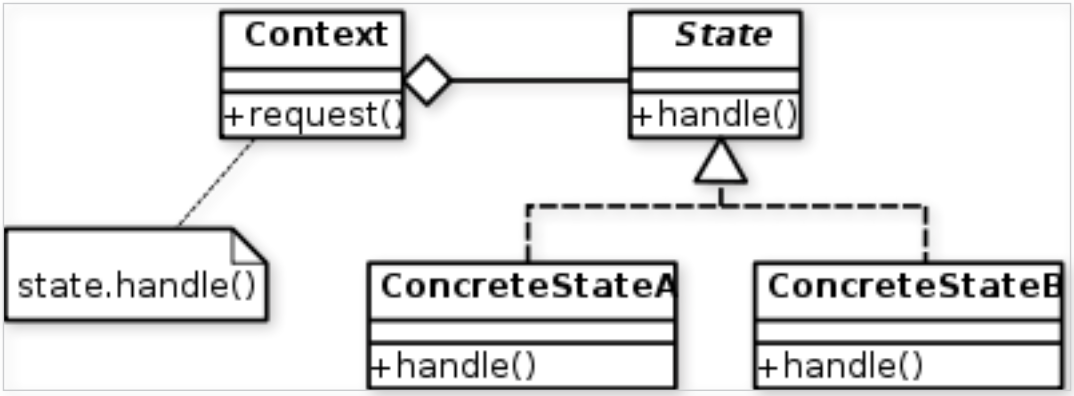

Fig.12. Diagramme UML pour le design pattern State

Ce patron montre les relations entre objets pour définir différents état tout en gardant la même structure depuis la source (l’objet `Context`). Il est donc possible d’avoir le même fonctionnement logique en utilisant un objet `ConcreteStateA` ou `ConcreteStateB`. La seule différence sera l’implémentation du code à l’intérieur de ces classes concrètes.

Les design patterns ne sont pas toujours une bonne solution selon le point de vue adopté. Par exemple, le Singleton pattern (voir Annexe 2) est souvent décrié comme un anti-pattern par beaucoup de programmeurs, car sa solution implique de rendre globales des informations dans le programme. On pourrait penser que ce n’est pas important qu’un objet soit global, mais tout les objets référencés par le singleton sont aussi rendu global, donc tout les objets de ces objets le sont aussi, et ainsi de suite. On obtient alors une quasi-infinité de variables rendues globales.

Bien que les design patterns s’occupent d’une organisation technique du code, il existe des concepts théoriques de design, afin d’orienter les développeurs dans leur conception.

2.2.3 Principles

Afin d'éviter au mieux les erreurs de conception et de développement, les ingénieurs se sont mis d'accords sur des «bonnes pratiques» en programmation. Ces pratiques sont définies pour les développeurs en tant que «guidelines», qu'il est vivement conseillé de suivre, et sont appelées des «principles».

Ces principles sont souvent sous des acronymes mémorables, et on retiendra notamment «Don't be STUPID : GRASP SOLID». Pour la cohérence de ce mémoire, nous n'allons pas étudier en profondeur ces principles, chacun étant très bien documenté sur beaucoup de sites internet. Vous retrouverez les descriptions basiques de ces trois principes en Annexe 3.

Si nous reprenons l'exemple du puzzle, les principles correspondraient aux conseils de bases de résolution du tableau, comme le fait de commencer par imbriquer les pièces situées sur le bord avant d'attaquer la partie centrale. Il n'est pas obligatoire de suivre ce conseil, mais il permet de continuer la progression du tableau en ayant une vision plus structurée du résultat final.

Certains principles s'appuient sur des design patterns pour orienter les programmeurs, tandis que d'autres définissent des moralités générales, comme par exemple d'éviter le «code spaghetti», ou de ne pas réinventer la roue carré (sous-entendu de ne pas réécrire du code déjà existant de moins bonne qualité). Certains s'occupent de la conception même d'un objet et de la manière dont il laisse ses informations disponible pour les autres objets.

Ces principles apportent donc une bibliothèque riche de conseils de mise en oeuvre d'un programme. En suivant ces directives, le programme développé a de forte chance d'être mieux compris par une majorité de développeurs, et d'être plus souple aux modifications à apporter.

En couplant les principles avec les design patterns, le système devient une structure plus solide et nous amène à concevoir des design plus complexes, à plus grande échelle, donnant naissance à une architecture complète.

2.3. L’architecture

Un programme orienté objet et donc composé d’un ensemble d’objets qui communiquent entre eux pour former un système complexe. L’imbrication de ces objets forme une cohérence, si bien que les possibilités d’assemblage sont quasi-infinies. Nous avons vu que les design patterns permettaient de répondre à des besoins spécifiques.

◊ Qu’en est-il du système dans son ensemble ?

Reprenons l’exemple du puzzle de la section 3.2 en changeant de point de vue. Admettons maintenant que le tableau entier représente le programme, et que chaque pièce correspond à un module ou un sous-sytème, lui même composé de plusieurs objets qui communiquent entre-eux.

Nous connaissons par expérience et par confiance les différentes formes disponibles dans la boîte. Il n’est donc qu’une question de temps et de volonté pour imbriquer les bonnes pièces ensemble et finir le tableau.

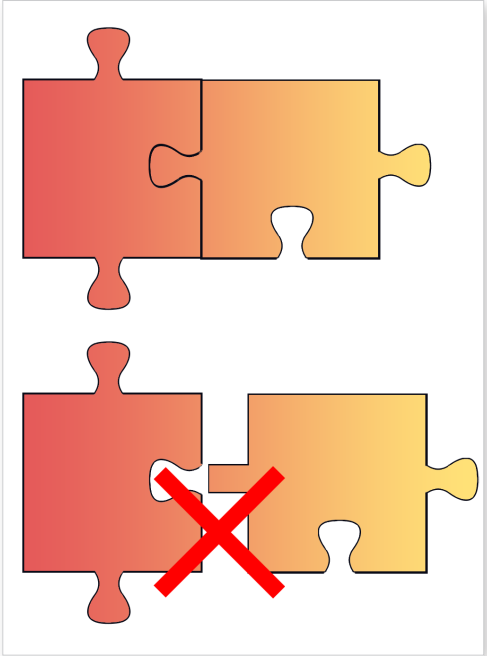

Fig.13. Représentation imagée d’une mauvaise conception en architecture

Nous savons qu’une pièce avec un bout sortant une tige arrondie rentre dans une pièce avec un creux arrondi. Cela nous parait normal. Que se passerait-il si les tiges arrondies soient remplacées par des tiges rectangulaires ? Nous n’aurions pas la possibilité de les imbriquer dans les creux, et nous auront alors un défaut de conception.

L’architecture du tableau du puzzle a été pensée pour que toute la structure soit cohérente et fiable. Il en est de même pour l’architecture d’un bâtiment. Remplacer un pilier porteur par une poutre décorative et l’édifice se brise.

L’*architecture logicielle* reprend les mêmes principes.

2.3.1 Quelques points importants

L’architecture établit principalement les liens entre les objets ainsi que la manière dont ils communiquent. Elle a pour objectif de rendre l’ensemble efficace dans le temps et de survivre à de nombreux changements. Afin de garantir la sécurité du système, elle organise généralement les différentes parties du programme dans un niveau hiérarchique sous formes de couches. Les couches les plus hautes dépendent généralement des couches inférieures et sont moins contraignantes pour l’utilisateur lambda.

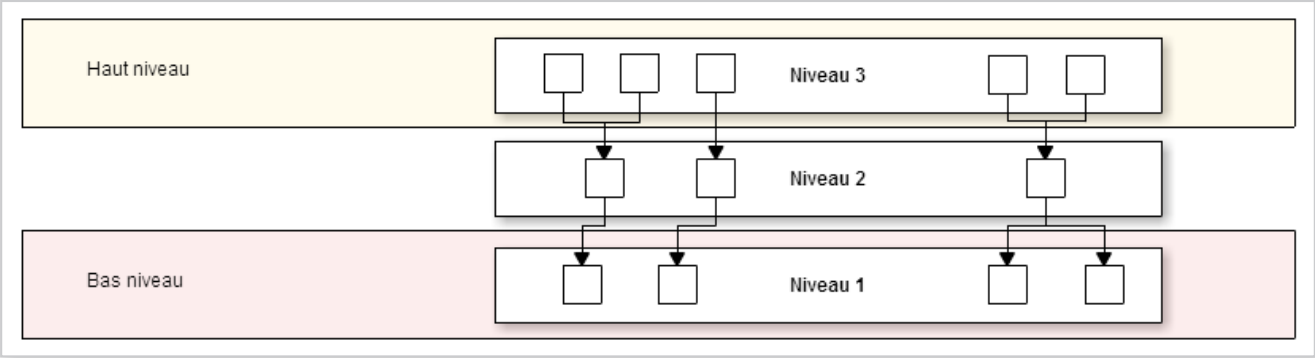

Fig.14. Schéma de présentation des différents niveaux de couches dans une architecture

Dans l’exemple de la maison, le particulier peut voir les murs peints, le sol carrelé, les prises électriques, qui correspondent à des éléments du «niveau haut» de la maison. Nous savons que des câbles électriques et la tuyauterie traversent les murs, afin d’alimenter la maison, mais ils ne sont pas directement exposés. Ils appartiennent donc à un niveau inférieur. Plus l’on descend dans les parties «cachées», plus l’on descend dans la hiérarchie de couches.

Il est de la responsabilité du concepteur de définir ces niveaux d’abstractions. Les programmes étant de plus en plus important avec le temps, cette tâche est devenu un métier à part entière : le métier d’architecte software.

L’architecte

L’architecte est en permanence en relation avec le chef de projet afin d’identifier les besoins, de designer une architecture, d’évaluer sa pertinence et les problèmes possibles, afin de minimiser les risques de production. Il est de la responsabilité de l’architecte de prévoir une extensibilité du système qu’il met en place et de garantir une stabilité à long terme. Il lui appartient aussi de déterminer quels frameworks , ensemble d’outils favorisant la conception de systèmes complexe, le projet a besoin pour développer l’application.

L’architecte doit avoir une vision globale du système à tout les niveaux de détails. Il doit être capable de designer les concepts fondamentaux et récurrents dans le système en général, mais aussi dans la composition des sous-systèmes.

Les outils de développement

Le développement d'une architecture requiert beaucoup de temps et de main d'oeuvre, et sa qualité dépend de l'investissement financier et temporel fourni par l'entreprise pour élaborer son projet.

Afin de faciliter le temps de production, l'équipe de développement peut être amenée à utiliser différents outils de développement. Parmi ceux-ci, nous retrouvons les langages de programmation utilisés ainsi que les bibliothèques de développement qui devront être choisis parmi les spécificités qu'ils peuvent apporter au projet.

Par exemple, dans le jeu vidéo, le C++ sera majoritairement utilisé pour sa rapidité d'exécution et sa gestion des ressources mémoires contrôlée pour des traitements de «bas niveau» qui sont indépendants du style de jeu représenté, comme le chargement de données, l'affichage de textures, les calculs d'intelligence artificiel, etc.

Pour les traitements de «haut niveau» comme les comportements de gameplay lié au jeu (ex : l'ouverture d'une porte), il sera préférable d'utiliser un langage plus convivial pour des développeurs parfois moins expérimentés, au prix d'une perte de rapidité dans l'exécution du programme.

Une fois les langages de programmation choisis, la production en programmation peut démarrer. Nous avons vu dans la section 3.2.3 sur les «principles» qu'il était déconseillé de réinventer l'existant. C'est pourquoi des entreprises ou des particuliers développent des fonctionnalités avancées par rapport à des besoins spécifiques. Ces fonctionnalités sont généralement sous la forme d'objets, regroupés ensemble pour former des bibliothèques, ou librairies, proposant des services, dont l'ergonomie et la facilité d'utilisation engendreront la réputation de celles-ci à travers le monde. Ces bibliothèques portent le nom de *framework* et sont capable d'être exportés pour d'autres solutions ou projets.

Nous pouvons par exemple citer le framework .Net de Microsoft, qui propose un ensemble conséquent d'outils pour la production d'applications interactives, notamment sur la plateforme Windows. Il existe des frameworks dans quasiment tous les domaines, que ce soit dans l'informatique graphique, sonore, l'imagerie médicale, le jeu vidéo, etc. Les architectures des programmes se basent généralement sur ces librairies pour diminuer le temps de production, néanmoins il est possible que l'entreprise décide de créer son propre framework si les besoins sont très spécifiques.

Un framework doit être capable de donner divers niveaux d'abstraction pour faciliter le développement. Ainsi, on n'accède plus par les composants «bas niveaux» qui sont utilisés pour «parler» à la machine, on utilise des objets répondant à la description logique de ses opérations.

Par exemple, il n'est pas utile pour un développeur de déterminer à chaque ouverture d'un fichier sur le disque dur quelles opérations doit faire le processeur pour obtenir les informations. Il est moins contraignant d'avoir un composant qui propose directement ces solutions. Le développeur n'a alors qu'à dire à ce composant «ouvre ce fichier» et d'obtenir le résultat voulu.

2.3.2 Styles d'architectures

De la même manière qu'il existe des styles architecturaux dans la conception d'édifice (style roman, baroque, etc), il existe différentes approches de design dans la conception logicielle. Le schéma suivant montre comment les modules de trois types d'architectures communiquent entre eux.

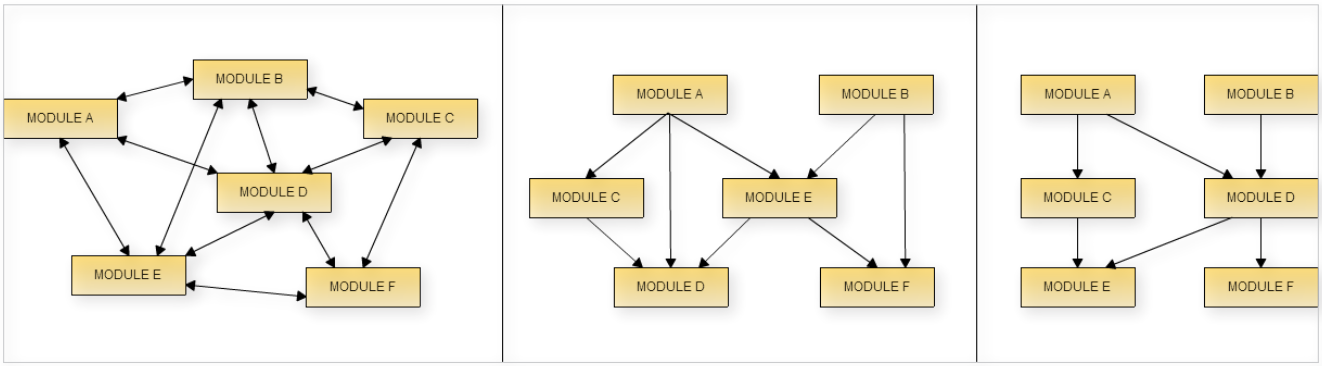

Fig.15. Schéma de présentation de différents style architecturaux d'un point de vue relationnel

◊ Le premier cas montre une architecture modulaire. Chaque module a accès à n'importe quel autre module. Ce type d'architecture est rapide à mettre en place, et est généralement utilisé lors du prototypage d'un système. Cependant, à mesure que le projet grandit, la maintenance devient de plus en plus complexe. La communication entre les modules devient serrée, et chaque module dépend d'un autre. Un changement important dans la structure d'un module implique des modifications dans tout ceux qui y sont liés. Ce cas est très rencontré lorsqu'aucune architecture n'a été réalisée en phase de conception du projet.

◊ Le deuxième schéma est une architecture en DAG (Directed Acyclic Graph). Elle a la particularité de contrôler les dépendances entre modules et d'éviter un «bouclage» entre ceux-ci. Si le module A dépend du module E, le module E ne pourra en aucun cas dépendre d'un module ayant un lien direct ou indirect avec le module A. Cela créée par la même occasion une hiérarchie dans les modules. Un module ne dépendra donc que de modules inférieurs à son niveau.

◊ Le troisième schéma représente une architecture par couches (ou layered architecture). Elle ressemble beaucoup à l'architecture en DAG, à la différence que chaque module d'une couche ne peut dépendre que des modules de la couche inférieure de degrés un. Si un module dépend d'une couche inférieur à un degrés supérieur, il y a alors une erreur de conception.

Certains types d'architectures sont aujourd'hui reconnus comme des modèles, on peut citer le «Service Oriented Architecture» (SOA), basé sur une composition de services, le «Layered Architecture», qui décompose le système en couches, où chaque couche opère par dessus une autre, ou bien encore le modèle «Client-Serveur», utilisé pour la transaction de données via un serveur web. Il existe une vingtaine de styles reconnus.

3. PROJET DE RECHERCHE INDIVIDUEL

Ma recherche tout au long de l'année s'est portée sur l'agencement de système dans un projet à moyenne ou large échelle, en étudiant les théories d'analyses comme les techniques mises en place dans un projet logiciel.

Il est évident que je ne pouvais pas produire un projet à large échelle par moi même, ni même programmer un système complet avec toutes les couches nécessaires au bon fonctionnement.

Dans le cadre du jeu vidéo, cela aurait impliqué la création d'un moteur de jeu capable de diriger les opérations liées à la machine, comme produire un rendu graphique 3D depuis une caméra, jouer un son, proposer des solutions d'IA, etc. Un travail qui aurait nécessité des années de développement avec une équipe dédiée.

Je me suis donc focalisé sur une couche d'abstraction supérieure, en utilisant comme base le moteur de jeu Unity3D. Nous avons évoqué au début de ce mémoire que dans le domaine du jeu vidéo, il y avait deux sortes de programmation : la programmation «engine» (liée au moteur de jeu et indépendante du jeu qui sera supporté dessus), et la programmation «gameplay» (code définissant tous les aspects techniques liés à un jeu précis, souvent non ré utilisé dans d'autres jeux).

Mon objectif était de travailler sur une partie «engine», en étendant les possibilités déjà offertes par Unity3D, et en utilisant les connaissances apportées par la recherche sur le développement software.

Ma motivation pour l'élaboration de cette extension fait suite à une conférence, «Unite 11», disponible gratuitement sur Internet, où des entreprises viennent discuter de leur pipeline de production sous Unity3D. Une des entreprises expliquait qu'ils avaient réussi à éclater leur production en divisant les parties programmation et graphique en différant projets, laissant plus de flexibilité dans ces deux domaines.

3.1. Harp Engine, extension de la plateforme Unity3D

3.1.1 Introduction

Motivations

Lorsque l'on utilise un moteur de jeu déjà commercialisé comme Unity3D et dont nous ne sommes pas propriétaire, nous sommes tenus d'utiliser les fonctionnalités que celui-ci nous apporte, et d'organiser notre programme par dessus.

Parfois, nous sommes tenus d'implémenter des fonctionnalités logicielle avancées pour supporter le projet. Nous retrouvons très fréquemment dans un jeu vidéo des menus, qui gère la configuration du jeu sur la plateforme utilisée par le joueur. Il lui est possible de modifier des paramètres graphiques ou sonores, afin de rendre le jeu plus compatible avec la puissance de sa machine, ou plus directement pour son expérience de jeu. Il peut aussi modifier les touches attribuées de son contrôleur type manette, ou clavier/souris pour adapter le jeu à ses habitudes. Ces informations doivent être stockées afin que l'utilisateur n'ai pas à les re configurer à chaque lancement du jeu.

Nous pouvons constater que ce type d'informations correspond à la globalité des jeux vidéos et pas seulement à un jeu spécifique.

◊ Serait-il alors possible de concevoir un système capable de gérer ces fonctionnalités indépendamment du jeu développé ?

Cette problématique m'a amené à réfléchir sur une plateforme indépendante, proposant des services aux développeurs afin de les aider lors de la production de leurs jeux. Cette plateforme sera disponible sous forme d'extension du moteur de jeu, sous le nom de Harp Engine.

Harp Engine

Harp Engine est le nom de l'extension que je met en place utilisant Unity3D. Unity3D propose tous les moteurs nécessaires au développement d'un jeu (graphique, physique, audio, ...). Cependant, il est à la responsabilité du développeur de les utiliser selon ses besoins. Le Harp Engine est une sous couche qui, parfois utilisant le moteur de jeu auquel il est associé, permet d'accroître les possibilités et le gain de temps lors du développement.

Unity3D possède des méthodes pour stocker de nouvelles informations, néanmoins le développeur n'a pas le contrôle absolu sur ces données. L'Harp Engine propose de rendre accessible au client (le joueur) ces données, par le biais de fichiers de configurations sur le disque dur, qui pourra les modifier sans passer par l'application s'il le souhaite.

C'est une pratique courante dans le jeux vidéo depuis des années de laisser des paramètres accessibles, dans des fichiers lisibles par un utilisateur (à l'inverse d'un fichier binaire lisible uniquement par la machine). Cela peut poser des problèmes de sécurité, mais il est à la charge du joueur de modifier ces fichiers que s'il n'a une parfaite connaissance des risques. Il est toutefois possible d'encoder les fichiers de configuration pour éviter une modification de ceux-ci.

Par exemple, il est possible de sauvegarder ou de récupérer des données très rapidement, comme par exemple activer/désactiver l'utilisation du sang dans le jeu (paramètre que le développeur aura lui même crée), qui affectera certains éléments du jeu, comme la création et affichage de «sprites» (plaques de textures dans l'espace 3D) de sang lors d'un combat. Ce paramètre pourra être disponible dans le menu du jeu pour que le joueur puisse activer/désactiver cette fonctionnalité.

3.1.2 Cahier des charges

L'Harp Engine est destiné à tout type de projet Unity3D de petite, moyenne et grande échelle. Son objectif est d'être le plus indépendant possible des spécificités gameplay du jeu en production. L'extension doit répondre à quelques critères afin de le rendre utile pour une communauté intéressée. Voici quelques points essentiels à son développement :

- ◊ Il doit pouvoir être utilisé par tout type de programmeurs, expérimentés ou non.
- ◊ Son accès doit être suffisamment simple et intuitif.
- ◊ Les traitements internes doivent être cachés de l'utilisateur.
- ◊ Les mis à jour de l'extension ne doivent pas pénaliser les versions antérieures.

3.2. Réflexion

Voici le schéma relationnel simplifié mis en place lors de la réflexion du projet. Au vue de l'utilisation de l'extension, j'ai opté pour une architecture en couches, où chaque couche repose sur une couche inférieure pour proposer ses traitements.

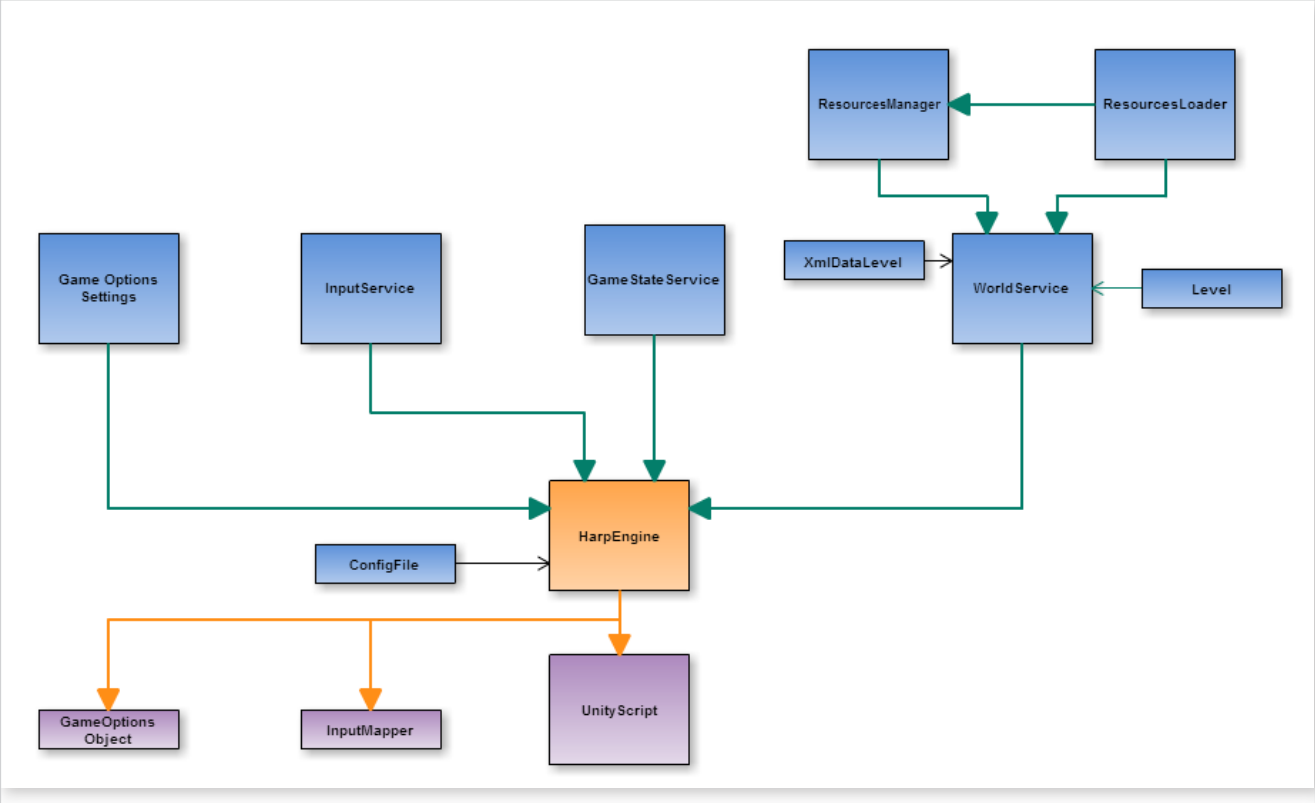

Fig.16. Schéma relationnel simplifié de l'extension Harp Engine

Nous pouvons déceler les différents niveaux de hiérarchie dans l'architecture. Les éléments les plus en bas sont les scripts Unity3D proposant une interface permettant de configurer l'Harp Engine. L'Harp Engine possède une plateforme globale qui permet d'accéder à ses services. Les éléments les plus au dessus sont des modules dont l'utilisateur n'aura pas accès, car ils sont spécifique au traitement interne de l'extension.

Abordons maintenant les services proposés par l'Harp Engine, en détaillant les motivations qui ont amenés à leur réflexion.

3.2.1 Services proposés

L'Harp Engine propose actuellement quatre services. Un service est similaire à un service dans la vie quotidienne. Un client demande un produit à un vendeur, ce dernier lui remet le résultat. Le client ne veut pas savoir comment son produit à été fabriqué, il veut simplement son achat. La transaction se fait parfois par le biais d'un médiateur, où le vendeur ne serait pas directement le responsable de la création de son produit, mais un tiers dont le client n'aurait pas connaissance.

Voici en détail la liste des services proposés par l'Harp Engine :

◊ **Game Options Service :** *Stocke les paramètres de l'application sur le disque dur.*

Unity3D dispose d'un système de sauvegarde de paramètres dans un fichier appelé «PlayerPrefs», mais ce fichier n'est accessible que par scripts. L'Harp Engine traite ces paramètres d'une autre manière, en sauvegardant les données dans un fichier externe lisible par un être humain.

Il donne aussi la possibilité aux développeurs de définir les paramètres directement dans l'interface d'Unity. Il devient alors possible de se détacher des seules options internes d'Unity.

Pour établir un lien direct avec le moteur, une classe est à renseigner par le développeur. (ex : assigner un paramètre d'AntiAliasing crée par le développeur au paramètre AntiAliasing d'Unity). On obtient un niveau d'abstraction supplémentaire par rapport au moteur Unity3D.

◊ **Input Mapper Service :** *Permet l'assignation dynamique des touches du clavier.*

Unity3D propose déjà un système d'assignation d'inputs, uniquement disponible en mode Editor. Il est donc impossible par l'API d'Unity3D de les modifier pendant le déroulement du jeu.

L'input mapper ré utilise quelques principes du système d'origine comme assigner les touches à des actions. Ce seront les actions qui seront utilisées dans l'application et non directement la touche.

Dans sa première version, l'input mapper ne supporte que le clavier comme type d'Input. Les contrôleurs de jeux type manettes n'étaient pas à ma disposition et n'ont donc pas été développés.

◊ **Game State Service :** *Gère l'état actuel de l'application.*

Ce service permet de déterminer dans quel état se trouve l'application (ex : InGame, MainMenu, etc). Il s'agit simplement d'une donnée modifiable par le développeur à n'importe quel moment. L'utilisation de ces états permet de définir des comportement différents dans l'application pour chaque. On peut par exemple empêcher l'utilisation de certaines touches du clavier lorsque l'on est en Menu Principal, ou alors modifier la vitesse du temps sur un autre état. Le développeur est responsable du comportement souhaité.

◊ **World Service :** *Établit un nouveau pipeline pour le chargement d'assets.*

Différentes méthodes existent sur Unity3D pour afficher des assets. On peut directement placer les objets dans une scène, et charger cette scène par script. L'inconvénient majeur de cette technique est que les assets doivent tous être compilés en même temps que le jeu lui même.

Si le projet devient large, le projet principal du jeu se retrouve extrêmement lourd, et il se peut que de nombreux crash surviennent. Cela oblige aussi les programmeurs et graphistes à travailler au même niveau dans la production. Celui qui est chargé de compiler l'application doit attendre d'avoir à la fois le travail des programmeurs et des graphistes. Le parallélisme dans le pipeline est restreint, notamment pendant la période de tests.

Il existe sur Unity3D un système de package d'assets appelés AssetBundles. Ces packages peuvent être chargés dynamiquement par scripts, c'est à dire pendant le temps d'exécution de l'application. Cette technique est souvent utilisés sur des applications smartphone pour alléger la taille de l'application qui sera stockée dans la mémoire du téléphone. Les packages sont ainsi téléchargés depuis un serveur web, stockés en mémoire, et vidés une fois l'application fermée.

Le World Service utilise ce système. Tout est regroupé en «scène» (différent de la scène Unity3D). Chaque scène est liée à un ou plusieurs packages que le développeur indique grâce à une interface. Ces packages seront crées par le ou les graphiste(s) du projet.

Ainsi, on reforme un monde à partir de packages qui ne sont pas présent dans le projet compilé, on a seulement édifié des liens. Il faudra cependant s'assurer que le package ciblé soit présent dans le dossier défini par l'Harp Engine.

Le nouveau pipeline se découpe ainsi :

◊ Création des AssetBundles, stockés dans un dossier prédéfini par l'Harp Engine.

◊ Définition des chemins d'accès pour l'Harp Engine via une interface sous Unity3D.

◊ Appel vers l'API de l'Harp Engine via un script pour charger une scène.

On obtient alors un découplage entre le projet destiné à la programmation pur de l'application et du projet maintenu par un graphiste qui compose son niveau.

3.2.2 Du point de vue utilisateur

L'utilisateur, que ce soit un designer ou un développeur gameplay, doit pouvoir utiliser l'extension le plus facilement et intuitif possible. Sur Unity3D, le paramétrage d'un projet peut se faire à l'aide d'interfaces utilisateurs directement dans le programme. Il est possible de placer dans sa scène un script qui va gérer l'ensemble de l'Harp Engine, c'est à dire le démarrer, le paramétrer, et permettre son accessibilité.

Cela se fait par le biais de scripts «MonoBehaviour», des composants reconnus par Unity3D proposant une interface de configuration. Les scripts MonoBehaviour servent de liens entre la logique du moteur de jeu comprise par les développeurs et le système de l'extension. En d'autres termes, les scripts «utilisent» les fonctionnalités disponibles de l'extension, de la même manière qu'un développeur lambda le fera lors de son développement. Ces scripts de base inclus dans l'extension permettent juste une approche «par défaut» de sa configuration.

Parmi ceux-ci, on retrouve les scripts permettant de renseigner les options graphiques, sonores, et gameplay du jeu, les options d'attributions de touches du clavier à une action (comme par exemple «avancer» pourrait être lié à la touche «Z»), ainsi que le paramétrage des packages d'assets déterminant les différents mondes dont le jeu a besoin.

Il n'y a donc plus de partie programmation pour la configuration de l'extension, et chaque projet est libre de définir ses options comme il le souhaite.

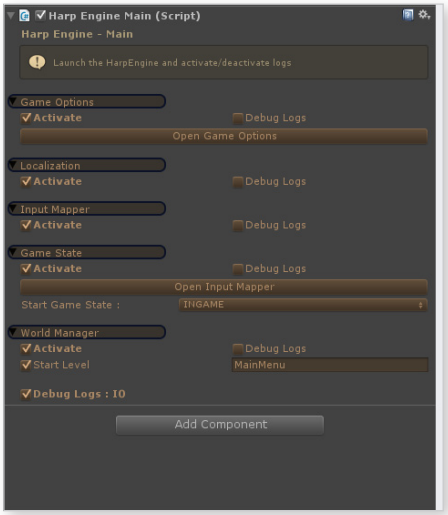

Fig.17. Exemple d'interface utilisateur conçue pour l'Harp Engine

Pour modifier ou récupérer une information, le développeur gameplay a à sa disposition la documentation nécessaire pour utiliser l'extension. Les traitements internes du système lui sont cachés, seul le résultat lui importe. Il peut donc facilement mettre en place ses interactions avec le système.

Voyons un peu plus en détail comment l'extension est mise en place.

3.3. Développement

3.3.1 Création de la plateforme principale

Dès le début du développement, une première problématique se pose.

Comment les développeurs vont utiliser cette extension ? Par quelle logique ?

L'extension est codée d'une certaine manière, mais le développeur lambda doit y faire appel à des moments voulu dans son propre code pour l'utiliser. Si la manière d'appeler les services de l'extension est trop compliquée, alors on peut se retrouver avec une mauvaise utilisation, des bugs, des plaintes, et bien plus encore. L'objectif est donc de rendre le tout ergonomique et simple d'accès, ne prenant pas en compte le niveau du développeur qui l'utilisera.

D'un point de vue plus technique, deux options s'offraient à moi :
Singleton ou Statique pur ?

Le Singleton est un design pattern permettant à une classe de n'avoir qu'une seule instance dans le programme, et d'être accessible depuis n'importe où... Ce qui correspond de manière très générique au comportement d'une classe statique pure. Mais le Singleton appartient plus à la méthode orientée objet que la classe statique.
Si l'on se réfère a *Misko Hevery*, ingénieur chez Google, le Singleton est un anti-pattern qui pose plus de problèmes qu'il n'en résout. De plus, sa syntaxe est moins aguicheuse qu'une classe statique.

J'ai donc pris le risque d'utiliser une classe statique comme «passerelle» au Harp Engine plutôt qu'un Singleton. Le Singleton n'étant peut être pas connu de tous, l'accessibilité du programme n'est donc pas menacée par ce choix.

3.3.2 Création d'un service

La particularité d'une extension est que les différents modules qui le composent peuvent être activés ou désactivés sans rompre le fonctionnement global. Chaque service est par défaut désactivé lors de l'installation du moteur. Il faut activer les services que l'on a besoin au démarrage de l'application. Cela permet de ne pas gaspiller de la mémoire inutilement.

Si le service n'est pas activé mais qu'une requête lui a été lancé, l'Harp Engine propose une solution pour éviter une erreur et un blocage de l'application. On utilise ici un **Null Service**. Le Null Service est une copie du service à la seule différence que les fonctions (méthodes) qu'il possède ne font rien. La requête passe donc bien où il faut mais l'information n'est pas traitée. Voir le design pattern Null Object en Annexe 2 pour plus d'informations.

Un service agît comme un tapis roulant dans un aéroport. L'utilisateur est le passager qui demande sa valise. L'employé de l'aéroport, quant à lui, est chargé de délivrer la valise sur le tapis qui sera retourné auprès du passager. Le passager n'a pas connaissance des actions de l'employé, il est obligé de lui faire confiance. L'aéroport est capable de délivrer tout types de valises dans une journée, et nous avons la le principe d'abstraction, qui définit l'objet dans sa vision globale et non dans son type précis. Ainsi, le tapis roulant marchera suivant que la valise soit petite, grosse, à poignée, etc.

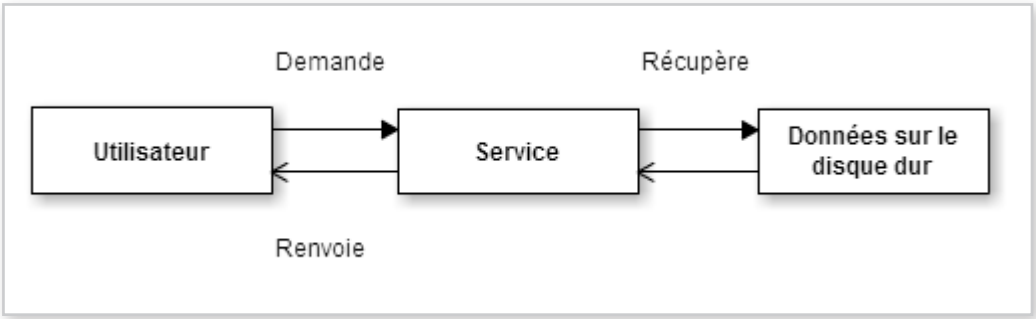

Fig.18. Représentation séquentielle d'une récupération de donnée sur disque dur

Dans notre cas, le service obtient ses «valises» depuis le disque dur où sont stockées les informations nécessaires pour l'utilisateur. Lorsque celui-ci les demande, le service est capable de retourner la valeur. Cela marche aussi dans l'autre sens, si l'utilisateur cherche à modifier une valeur.

3.3.3 Création des interfaces utilisateur

Comme l'extension est incluse dans le moteur Unity3D, et que le moteur propose un système d'interfaces utilisateur accessible et programmable, il était naturel de proposer toute sorte d'interfaces sous formes de fenêtres afin de paramétrer le plus simplement possible l'Harp Engine.

Afin de garder une cohérence graphique dans ces fenêtres, j'ai décidé de programmer en premier lieu le coeur de toutes les fenêtres dans une classe abstraite, dont toutes les interfaces hériteront par la suite. Ainsi je peux paramétrer une seule fois les couleurs utilisées, les tailles des étiquettes, etc., chaque fenêtre se mettant à jour au moindre changement.

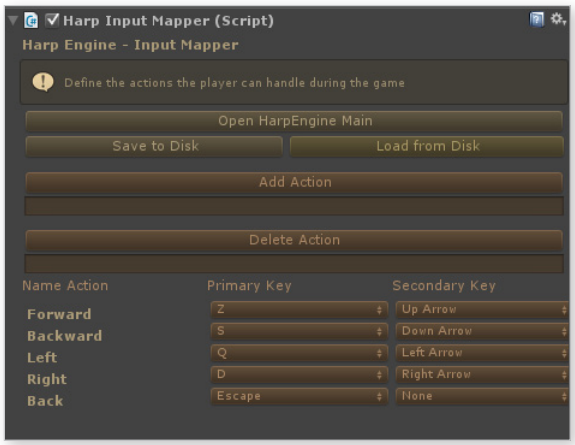

Fig.19. Exemple d'interface conçue pour l'Harp Engine

Les données que l'utilisateur inscrits en utilisant ces interfaces s'enregistrent sur le disque dur dans les fichiers de configurations. Il est toutefois possible de modifier les valeurs à la main en ouvrant directement le fichier en question, mais il est préférable d'utiliser les interfaces pour plus de sécurité.

```
1  <root>
2    <GameOptions>
3      <width value="1280" />
4      <height value="720" />
5      <Brightness value="3" />
6      <Fullscreen value="1" />
7      <vSync value="0" />
8      <AntiAliasing value="0" />
9      <AnisotropicFiltering value="2" />
10     <TextureQuality value="1" />
11     <VolumeGlobal value="10" />
12     <VolumeSounds value="10" />
13     <VolumeVoices value="10" />
14     <VolumeMusic value="10" />
15     <Language value="0" />
16     <SensitivityX value="5" />
17     <SensitivityY value="5" />
18   </GameOptions>
19 </root>
Ln : 19  Col : 8  Sel : 0  Dos\Windows  ANSI
```

Fig.20. Données utilisateurs sauvegardées sur disque dur dans un fichier XML

3.4. Conclusion

3.4.1 Utilisation de l'extension dans un projet

Au mois de Janvier 2014, nous avions pour objectif de former un groupe d'étudiants et de mélanger les diverses compétences théoriques et techniques acquises de chacun dans un seul et même projet, sur une durée de trois semaines.

Mon groupe s'est composé de Jérémie Anziani, dont la recherche se portait sur l'optimisation graphique dans le jeu vidéo, ainsi que de Simohamed Ouarch, travaillant sur du rig quadrupède.

Notre jeu, intitulé Wattelez, est un survival angoissant se déroulant en pleine nuit dans une usine désaffectée. Le joueur est muni d'une torche enflammée qu'il doit réapprovisionner sur des barils de feu disséminés dans la carte. Le joueur fait face à une horde de chiens sauvages qu'il doit faire fuir avec sa torche pour survivre.

Il n'y a pas de réel objectif dans cette démonstration. Nous voulions avant tout proposer des techniques de production afin de rendre le jeu extensible rapidement.

Afin de tester les capacités de l'extension Harp Engine que je développais en parallèle, nous avons décidé d'utiliser le pipeline spécifique à l'instanciation dynamique des assets 3D, en découplant le projet «programmation» du projet «graphique». Nous avons donc utilisé en premier lieu le World Service.

Cela nous a permis de vite identifier les bugs et les modifications nécessaires pour obtenir un workflow efficace dès les premiers jours. Une fois le service opérationnel, l'équipe était capable de travailler sur sa partie dédiée sans se soucier de l'avancement de ses collègues. Le graphiste était donc capable de travailler indépendamment sur des scènes 3D entières, qu'il exportait en AssetBundle, et qui étaient chargées dynamiquement par le code après compilation du projet. Il n'y avait plus besoin de transférer les assets d'un projet à un autre et de recompiler le jeu pour observer les nouveaux graphismes.

De plus, il m'était possible d'effectuer tous les tests de gameplay dans une scène de tests en attendant une version alpha du terrain de jeu principal.

Celà nous a permis un gain de temps non négligeable sur la suite de la production.

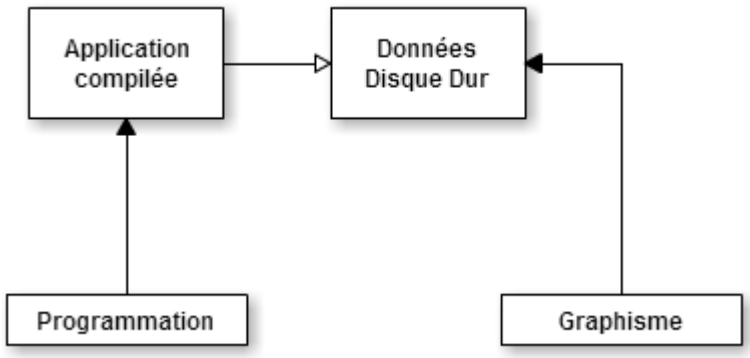

Fig.21. Le partie programmation découplée de la partie graphisme

Durant la phase de production graphique, dans les premiers jours, j'ai pu me concentrer sur l'élaboration du menu principal utilisant le GameOptions Service de l'Harp Engine.

La structure même du menu est inspirée de la librairie NGUI (disponible sur l'AssetStore), bien qu'elle soit très incomplète comparée à cette dernière.

Sur les quelques projets Unity3D que j'ai pu développer avant celui-ci, les menus que j'avais mis en place étaient suffisant pour le jeu auquel il était dédié, mais il me manquait une couche plus «générale» me permettant de les réutiliser dans d'autres projets. J'ai donc pris la décision de re-coder entièrement le système de menu.

Grâce aux design patterns, j'ai pu édifier un système plus extensible et plus flexible que je ne l'aurais fais sans. Chaque élément du menu est un objet qui répond à un parent. A chaque opération demandée par un parent, un message est envoyé à tout ses enfants de s'exécuter. Un compteur d'opérations est présent pour déterminer combien de tâches sont en cours. Chaque fois qu'un enfant a fini une tâche, il renvoie un message à son parent pour lui indiquer qu'il a finit.

Grâce à la hiérarchie installée, tout le système est maîtrisé. Il m'était possible de bloquer des interactions lorsqu'une opération était en cours, comme le fait de ne pas pouvoir cliquer sur un bouton qui était en train de disparaître dans le temps.

Fig.22. Menu principal du jeu utilisant l'Harp Engine

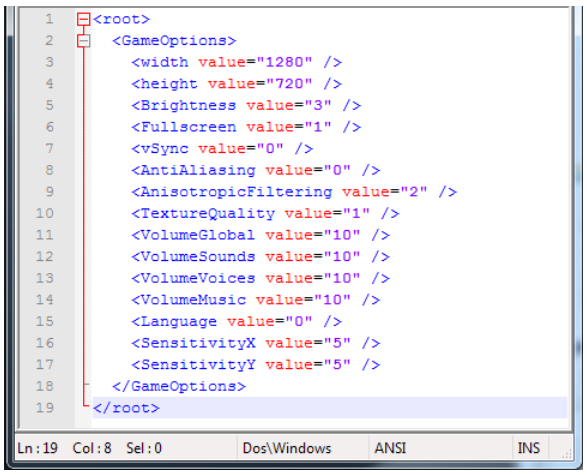

Fig.23. Données sur disque dur récupérée par l'Harp Engine

Pour afficher des données, les scripts du menus accèdent directement à l'Harp Engine par le «GameOptions Service», et sont capables de les modifier grâce à l'interface proposée par le service.

La création du coeur du système de menu a pris environ deux jours, avec un jour de plus pour la finalisation et le débug.

Au fur et a mesure que le projet avançait, de nouveaux paramètres étaient nécessaires, il fallait donc les rajouter dans le menu. Grâce au système mis en place, l'ajout de paramètres ne prenait que quelques minutes.

3.4.2 Forces et faiblesses de l'extension

L'Harp Engine permet d'exploiter le moteur de jeu pour développer plus facilement des jeux vidéos, sans repartir de zéro à chaque nouveau projet. L'extension est configurable par projet, si bien qu'aucune donnée concrète n'est utilisée dans le code. On pourrait voir l'extension comme un lave-vaisselle. Le lave-vaisselle possède un fonctionnement interne, et propose à ses utilisateurs d'insérer ses données (la vaisselle) dans des compartiments alloués. Une fois le traitement fini, les données sont retournées dans un état modifié. L'utilisateur peut placer tout type de données dans la mesure qu'elles soient compatibles avec le système.

L'Harp Engine stocke les données utilisateurs sur le disque dur, elles ne sont pas perdues et sont réutilisables lors d'une réutilisation de l'application. Pour l'instant, l'extension ne propose qu'une sauvegarde sur un ordinateur de bureau. Le stockage des données étant différent sur des appareils mobile ou des plateformes web, l'Harp Engine n'est pas encore compatible avec ces périphériques.

Bien qu'il soit possible de configurer dynamiquement les assignements de touches du clavier à des actions, il n'est pas encore possible d'utiliser d'autres périphériques tels que des manettes de jeux ou autres types de contrôleurs.

Le World Service permet d'accéder à un autre type de pipeline pour le chargements dynamique d'assets. Cependant, sa version initiale a été développée pour le projet Wattelez décrit plus haut, et n'est pas encore suffisamment paramétrable pour obtenir différents comportements. A l'heure actuelle, un package est téléchargé puis directement instancié dans la scène. Il est envisageable que l'on veuille télécharger un package et stocker en mémoire les assets pour une utilisation future, ou une réutilisation.

4. CAS CONCRET EN ENTREPRISE

D'octobre 2013 à mai 2014, j'ai travaillé à la Société Orbe, basé à Paris. Cette petite structure produit des applications mobiles de réalité augmentée dans divers domaines : reconstitutions historiques, reconstitutions d'un environnement sonore, conception et développement de guides mobiles de lieux culturels tels que des bibliothèques ou musées, en utilisant diverses technologies tel que la géolocalisation ou l'orientation axée sur le gyroscope et la boussole interne du périphérique mobile.

Afin de représenter le monde en trois dimensions, l'entreprise utilise Unity3D comme moteur temps réel. Bien que ce ne soit pas directement du jeu vidéo, les applications sont gérées d'une manière similaire (chargement de scènes, de modèles 3D, de textures, etc...) en y ajoutant une dimension logicielle.

Le projet auquel j'ai été assigné tout au long de l'année s'intitule «Musée d'Histoire de Marseille» (MHM), une application mobile destinée à compléter les visites guidées du musée dans les lieux historiques de la ville à travers différents types de contenu. L'utilisateur est positionné sur une carte 3D en vue aérienne, géolocalisé grâce au GPS du périphérique, et oriente sa vue suivant les points cardinaux.

A différents endroits clés de la ville, représenté par des pastilles sur la carte, l'utilisateur a la possibilité d'accéder à du contenu : on retrouve notamment des vidéos ou des témoignages d'experts archéologues, ou bien des reconstitutions en trois dimensions stylisées des environnements d'une autre époque.

Tout les contenus sont stockés sur un serveur web, ils sont ensuite téléchargés, traités, et affichés dans l'application.

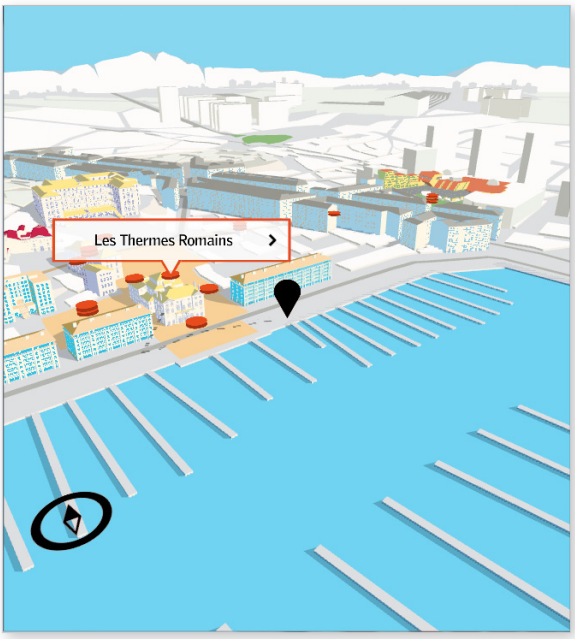

Fig.24. Capture d'écran de l'application mobile Musée d'Histoire de Marseille, Société Orbe, Paris

Le code produit dans ce projet est destiné à être réutilisé dans des projets futurs similaires. Il a donc été question d'un *refactoring* complet du code, c'est à dire sa réorganisation et sa restructuration, afin de le rendre plus flexible et maintenable. L'objectif est de faire gagner du temps à l'entreprise pour configurer ses projets à partir d'une plateforme unique, et d'aider les futurs développeurs à identifier et réparer les bugs rapidement.

La particularité de ce refactoring est de permettre à un designer d'avoir la possibilité de paramétrer son projet et les comportements liés à divers systèmes, directement dans l'interface d'Unity3D, en laissant la responsabilité des traitements au framework qui agit d'un point de vue abstrait. Ainsi, aucun comportement concret n'est défini dans le framework, seulement une suite logique d'opérations.

4.1. Développement d'un nouveau framework

Lorsque l'on travaille sur un projet existant, il faut faire très attention aux modifications que l'on opère dans le code. Dans un projet comme MHM, on compte plus d'une cinquantaine de classes et des milliers de lignes de code pour supporter l'application.

Mes premières tâches étaient de déboguer quelques fonctionnalités ainsi que d'en rajouter.

En parcourant les différentes classes liées au problème en cours, je me suis très vite aperçu d'une chose : la quasi totalité du code était en *programmation procédurale*, soit une succession d'événements contenues dans des classes et fonctions statiques appelées plusieurs fois dans d'autres classes.

De plus, aucune hiérarchie n'était visible, si bien que toutes les parties du code se retrouvaient mélangées entre elles, créant ainsi des dépendances bien trop importantes.

La décision a été prise de créer un framework suffisamment générique pour supporter les types d'applications que l'entreprise produit à partir du code existant sur «MHM». La réflexion principale se portait sur le passage en programmation orienté objet du système. De cette manière, les modules et comportements de l'application doivent au maximum éviter le couplage sérré avec d'autres modules pour les tenir indépendants sur le long terme.

En utilisant des *design patterns* et en suivant les *principles* définis dans la première partie de ce mémoire, il était possible de réduire les conséquences de dépendances en modifiant certaines logiques.

4.1.1 A partir de l'existant

Dans un développement d'application ou logiciel, il intervient plusieurs étapes de production. Ces étapes sont généralement dues au temps de développement alloué, ainsi qu'au coût de production.

Lorsqu'un client demande à un prestataire de lui développer une application originale, il est fréquent que les développeurs travaillent rapidement sur une version prototype afin d'identifier les problèmes et les solutions à venir. Cela génère dans la plupart des cas du code non hiérarchisé, non structuré. Les éléments principaux sont écrits mais le design du code n'est pas forcément établi. Ce prototype permet d'établir un lien avec le client très rapidement.

Lorsque le prototype est validé, le développement devient plus concret : des fonctionnalités sont ajoutées, et souvent l'équipe de développement fait appel à d'autres développeurs pour enrichir la main d'oeuvre. Cet ajout a parfois pour conséquence de rendre le code moins unifié, car il provient de différentes sources de développement, ayant une culture et une approche différente de la programmation.

Si l'on admet qu'aucune décision de design et de «guidelines» ne soient prises dès le départ, le code résultant se retrouve très diversifié, et l'on obtient dans la plupart des cas ce que l'on appelle du *code spaghetti* (voir section 3.2.1) . La métaphore provient du fait que le code est tellement déstructuré, que l'ensemble du projet ressemble à un plat de spaghettis, toutes les parties se mélangeant les unes aux autres.

Par exemple, il se peut qu'à la suite d'un développement à long terme sur un module, travaillé par plusieurs programmeurs qui ne se sont peut être jamais rencontrés, quelques fonctions soient doublées car l'un des programmeurs n'aura pas vu que cette fonction existait ailleurs dans le code.

C'est alors qu'intervient le concept de *refactoring*. Lorsque le développement a atteint un stade mature et est utilisé sur une application concrète déjà commercialisée ou disponible pour le public, la décision peut être prise de rendre le code plus optimisé et surtout réutilisable dans des projets futurs. La réorganisation du code est primordiale, et il est nécessaire d'avoir une vision globale de l'ensemble du projet avant de prendre des décisions de refactorisation.

◊ Quels aspects sont redondants ? Comment les modules communiquent entre eux ? Comment les objets sont définis pour traiter les informations ?

Ces questions s'appliquent à tout les niveaux de détails du projet. Pour préparer tout le travail d'architecture et de refactorisation, il est conseillé d'élaborer les schémas UML du projet existant si ce n'est pas déjà fait, et d'en déceler toutes les parties qui peuvent être regroupées. De plus, il sera possible d'identifier les dépendances entre chaque classe du projet. A partir de là, la réorganisation du code peut se faire petit à petit, en s'assurant qu'un changement n'impacte pas de conséquences dans l'exécution du programme.

Dans le cas du projet MHM, nous nous étions retrouvés avec une architecture de la forme suivante :

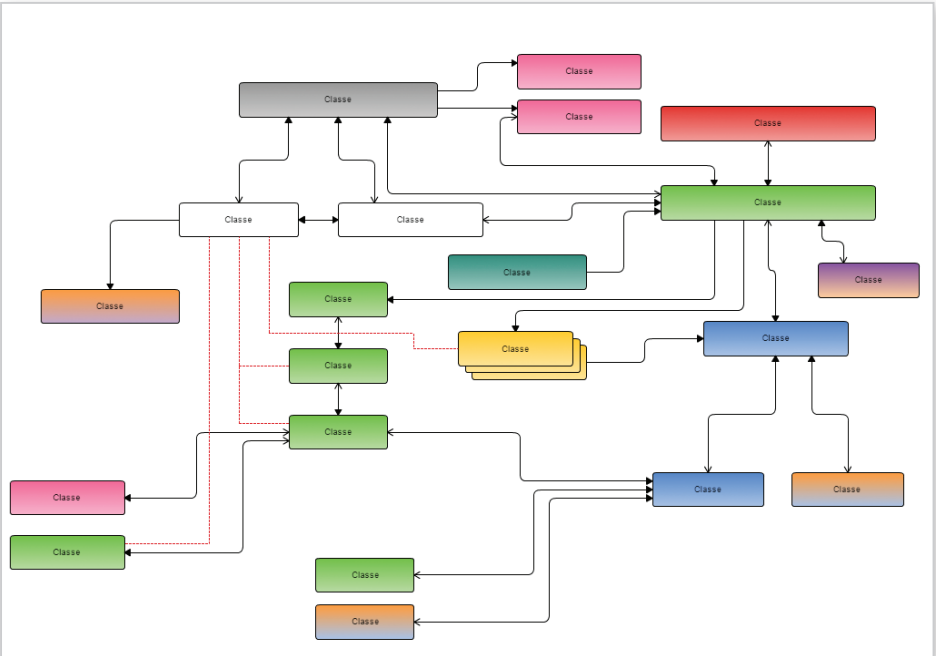

Fig.25. Schéma UML simplifié de l'architecture existante sur le projet MHM, Société Orbe, Paris

4.1.2 Designing d'une architecture

La réflexion sur le design que nous allons étudier est strictement personnelle et découle des problèmes rencontrés dans le code existant. Il ne s'agit pas forcément d'un «bon design», mais d'une solution.

La première partie d'analyse du refactoring a été focalisée sur le regroupement de classes par domaine d'application dans le programme (représenté par les couleurs sur le schéma ci dessus). Par exemple, tout ce qui était lié au comportement de la caméra dans l'espace tridimensionnel était regroupé dans une section «Camera». Nous avons ainsi pu déterminer quels types d'activités le programme requérait pour son fonctionnement.

Si nous reprenons le schéma complet de l'architecture du programme, nous obtenons désormais un design ressemblant à celui-ci :

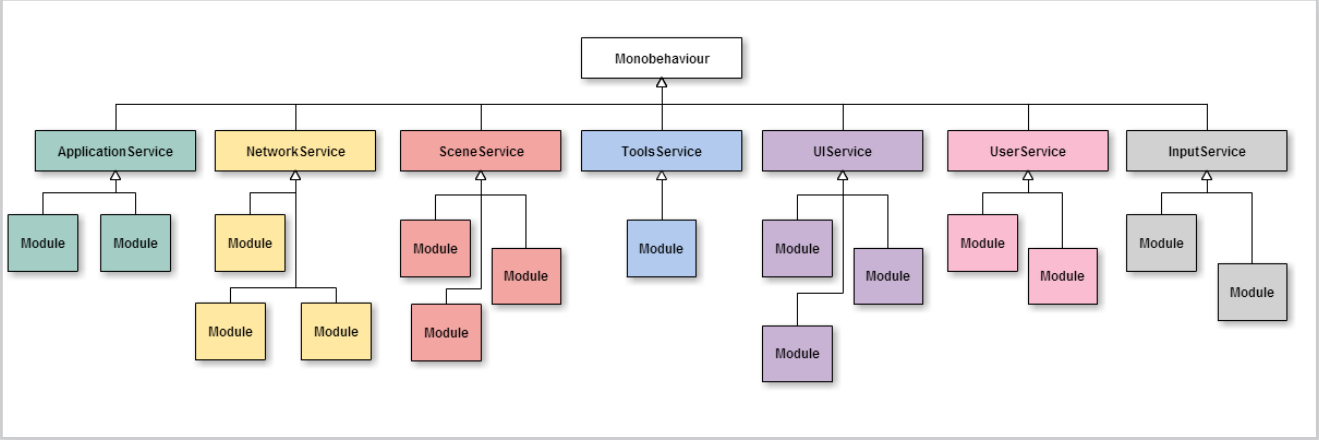

Fig.26. Schéma simplifié de la nouvelle architecture réfléchie, Société Orbe, Paris

Nous avons évoqué dans une autre partie que le code existant était majoritairement composé de programmation procédurale, au moyen de classes statiques[1], créant des données globales dans tout le programme (voir section 3.1.2, sur la programmation procédurale). Comme le programme repose sur cette structure, il était quasi impossible de casser ce fonctionnement. Cependant, nous pouvions réfléchir à un moyen de réutiliser le code dans une optique plus orientée-objet. Pour cela, tous les objets d'un domaine ont été regroupé.

1 A l'inverse des classes objets, une classe statique est disponible dans l'ensemble du programme. Elles sont généralement utilisées pour proposer des traitements génériques, comme par exemple des conversions. L'utilisation de classe statique est déconseillée pour du traitement spécifique lorsque le programme est orienté-objet.

Ces objets ont ensuite été découpé pour former des modules indépendants. Un module est donc un ensemble d'objets liés à une même problématique, et se caractérise par une partie accessible depuis l'ensemble du programme, et une partie non accessible, donc il est le seul maître. L'ajout de module apporte de la flexibilité au programme, car il est possible de remplacer leur fonctionnement rapidement sans dédommager le reste de l'application.

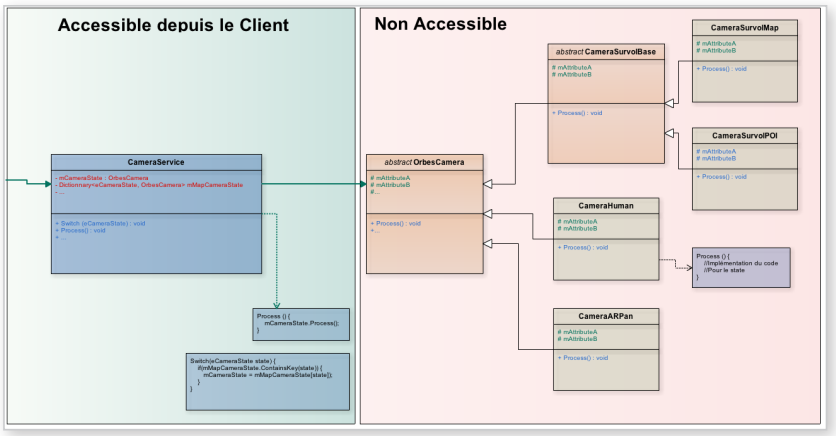

Fig.27. Représentation d'un module à partir d'une composition par abstraction, Société Orbe, Paris

Il arrive fréquemment qu'un module traite une information pouvant avoir différents comportements suivant l'état dans lequel il se trouve. Un comportement détermine la capacité d'une entité à réagir face à une demande. Si l'on prend l'exemple de la caméra, nous pouvons lui attribuer un comportement de zoom. Cela ne nous dis pas qu'elle sera sa vitesse de zoom, simplement qu'il en est capable. La vitesse de zoom pourra être gérée directement dans le code, mais les possibilités de configurations deviendront limitées, chaque projet utilisant la même valeur de zoom.

Qu'en est-il si nous voulons paramétrer chaque comportement indépendamment, comme par exemple gérer la valeur de vitesse de zoom ? Qu'en est-il si un même comportement est partagé sur deux projets différents mais avec des valeurs différentes ? Nous voulons laisser à la disposition du designer le paramétrage de cette valeur, grâce à une interface Unity3D.

4.2. Au service du designer

Une des solutions adoptée a été de procéder par une composition par abstraction, basé sur le design pattern State (Annexe 2). Chaque comportement est stocké dans un objet répondant à une classe abstraite, qui l'implémente selon ses besoins. On a donc autant d'objets hérités que de comportements. Pour les rendre disponible dans l'interface d'Unity3D, ces objets sont des MonoBehaviour, les scripts reconnus par le moteur. De ce fait, leur paramétrage est possible comme sur le schéma ci-contre.

Nous avons désormais un système reposant sur des couches d'abstractions, n'attendant que les objets concrets implémentant le code nécessaire aux traitements et les données personnalisables définies par le designer du projet.

Le schéma suivant montre les différents niveaux de responsabilité dans le système :

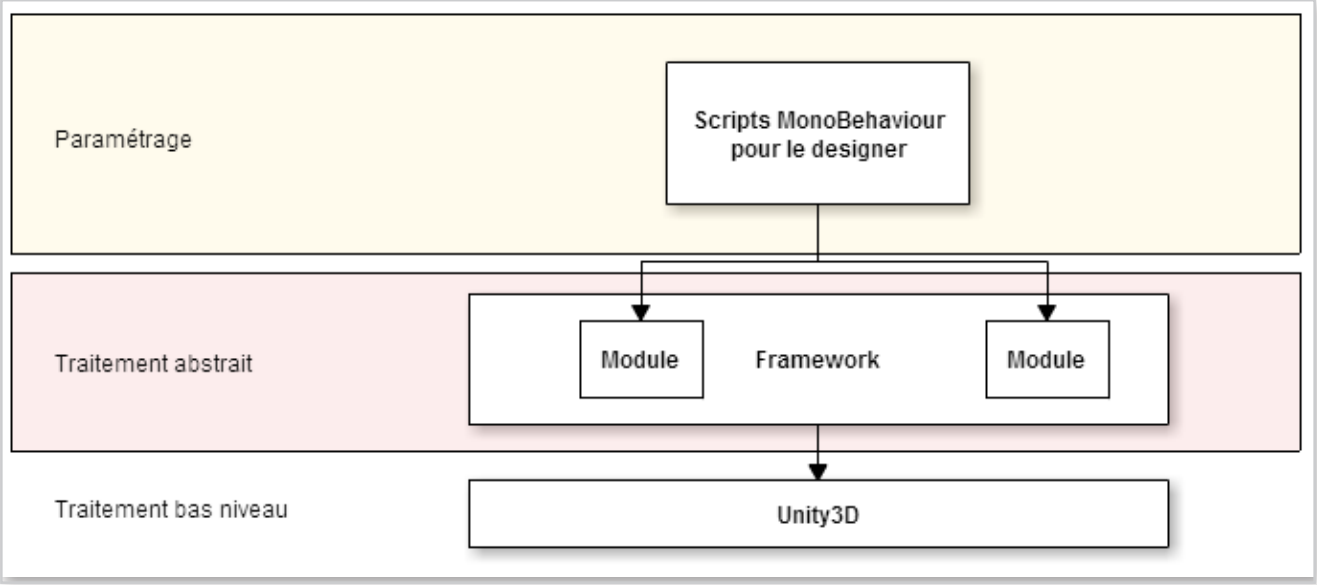

Fig.28. Schéma des différentes couches d'abstractions pour le projet MHM, Société Orbe, Paris

Les comportements peuvent être pris en charge par un développeur moins expérimenté, sans qu'il n'ait besoin de modifier le module qu'il traite. Le module se charge de récupérer un type d'objet défini avec une certaine interface, et il est de la responsabilité du développeur de respecter cette interface pour que son code puisse fonctionner.

D'un autre côté, les interfaces utilisateurs d'Unity3D permettent au designer de choisir les comportements souhaités pour son projet, d'en désactiver s'il le souhaite, et de renseigner les valeurs nécessaires. Dans l'exemple des scripts liés à la caméra, il lui est possible de déterminer la valeur de zoom maximum et minimum pour chaque comportement de caméra.

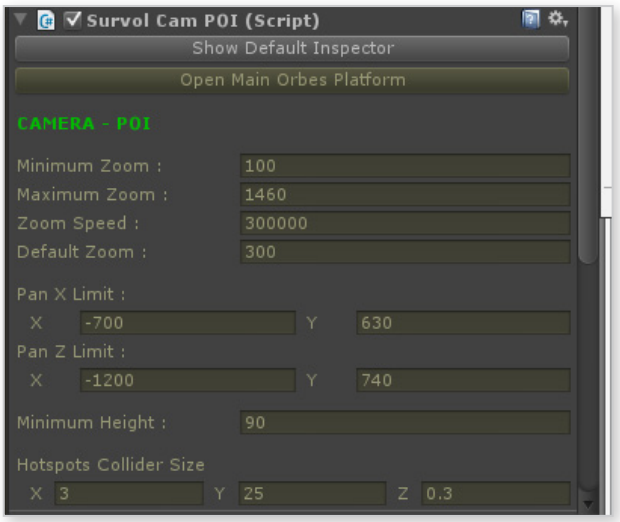

Fig.29. Exemple d'un script Monobehaviour pour designers dans le projet MHM, Société Orbe, Paris

Le framework lui même est paramétrable, et permet d'activer ou de désactiver des fonctionnalités principales.

Par exemple, si un développeur travaille sur une partie traitant de la géolocalisation de l'utilisateur dans l'espace 3D, il peut très bien désactiver le chargement d'assets 3D, affichant les décors, qui ne lui sont d'aucune utilité. Cela permet aussi de s'assurer que les modules sont suffisamment indépendants pour être traités séparément.

Nous avons donc rempli notre objectif en rendant les outils paramétrables et plus facilement extensible, permettant de réutiliser le framework dans d'autres applications du même type.

Fig.30. Plateforme principale de configuration du framework, Société Orbes, Paris

5. CONCLUSION

Nous avons étudié à travers ce mémoire l'importance de l'architecture logicielle dans un système informatique. En utilisant les techniques mises à notre disposition, comme les design patterns ou les principles, nous sommes capable d'édifier des structures solides tout en réduisant les coûts de production ainsi que le temps de développement. Il faut garder à l'esprit qui sera la cible des outils que nous développons. Que ce soit des artistes, des designers, ou des développeurs, le système conçu doit être suffisamment clair et intuitif pour leur durée de vie.

Mon projet de recherche personnel, l'Harp Engine, a permis un gain de temps considérable dans la production à court terme d'un jeu vidéo en équipe, et il m'est possible de réutiliser cette solution dans des projets futurs tout en le maintenant à jour. En parallèle, la refonte de l'architecture d'un projet au sein de la société Orbe diminue l'investissement financier nécessaire au développement de nouvelles fonctionnalités et du paramétrage global des applications qu'elle déploie.

Il ne faut toutefois pas oublier que la programmation est un domaine évolutif. Les problématiques rencontrées tous les jours amènent les développeurs du monde entier à se mobiliser sur des solutions, afin de rendre les produits de meilleure qualité au fil des années. La programmation structurée a amenée des concepts théoriques révolutionnant la manière d'implémenter le code au sein d'un programme, et la programmation orienté-objet a changé la vision d'un système informatique, en s'approchant au plus près du raisonnement humain.

Avec l'avancée technologique matérielle et les processeurs multi-coeurs apparus depuis quelques années, la programmation s'est étendue vers du traitement non plus séquentiel, mais parallèle, supportant plusieurs traitements pouvant se réaliser en même temps. Il est possible grâce à des architectures complexes de disperser des calculs très lourds à travers Internet pour que des milliers de machines participent aux traitements nécessaires. Il est donc important pour un programmeur de s'adapter aux nouvelles technologies, que ce soit matériel ou logicielle, et de comprendre les besoins qui ont amenés à cette évolution. La programmation dans quelques années ou décennies pourrait ne plus ressembler à celle que l'on connaît aujourd'hui.

6. GLOSSAIRE

Attributs 10 : Entité qui définit une propriété d'un objet. Il possède un nom et fait référence à une valeur, qui peut être une objet.

Dépendance 8 : Relation entre deux classes qui ont besoin de connaître l'existence de l'autre pour fonctionner. Une trop forte dépendance limite la réutilisation d'un objet dans un autre projet.

Encapsulation 11 : Principe élémentaire de l'orienté objet qui vise à ne rendre accessible que les données nécessaire à un traitement extérieur à l'objet lui même.

Framework 19, 34 : Ensemble d'outils de développement sous forme de bibliothèque, regroupant de nombreuses fonctionnalités pour diminuer le coût de production et les temps de développement.

Méthodes 10, 22, 24 : Fonction en programmation orienté objet caractérisant une action, effectuant des modification d'état de l'objet auquel il est associé. Une méthode peut être publique, privée ou protégée suivant son utilisation.

Objet 10 : Entité principale dans la programmation orientée objet qui a pour but de retranscrire des objets de la vie réelle en valeur informatique.

Procédures 8 : Fonction en programmation procédurale implémentant une suite d'instructions.
Programmation orientée objet 10 : Concept de programmation permettant de retranscrire au plus proche les comportements et le raisonnement humain. Son expansion a permis le développement de design et d'architecture logicielle. Il est le mode de programmation le plus utilisé de nos jours.
Programmation procédurale 8, 10 : Concept de programmation permettant de délivrer des instructions au processeur, l'unité centrale de traitement de la machine. Ce concept est de moins en moins utilisé à cause de sa difficulté de compréhension et de maintenance.
Programmation structurée 5, 6, 40 : Concept de programmation sous forme de guide révolutionnant les principales syntaxes à utiliser lors de la conception d'un programme.

Refactoring 48 : Restructuration du code après maturité pour gagner en qualité, en efficacité, en coût de maintenance et permettre une réutilisation du code pour des futurs projets.

Variable globale 8 : Entité représentant une valeur ou un objet le rendant disponible dans l'espace global du programme, accessible depuis n'importe où. L'utilisation d'une variable globale peut entraîner des problèmes de maintenance à large échelle.

7. BIBLIOGRAPHIE

Bersini, H. (2013) La programmation orientée objet (6ème Edition)

Cwalina, K., Abrams, B. (2008) Framework Design Guidelines (Second Edition)

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994) Design Patterns : Elements of Resusable Object-Oriented Software

Rabin, S. (2009) Introduction to Game Development (Second Edition)

8. TABLE DES ILLUSTRATIONS

Annexe 1

```
public abstract class MoteurFactory {
   public abstract Moteur GetMoteur();
}

//Sous classe de MoteurFactory
public class MoteurRenaultFactory : MoteurFactory {
   public override Moteur GetMoteur(){
        return new MoteurRenault();
   }
}

//Sous classe de MoteurFactory
public class MoteurCitroenFactory : MoteurFactory {
   public override Moteur GetMoteur(){
        return new MoteurCitroen();
   }
}


public class Voiture {
   public Voiture(MoteurFactory moteurFact){
        Moteur monMoteur = moteurFact.GetMoteur();
   }
}

//Depuis une partie extérieur du code
//qui instancie un objet de type Voiture
MoteurFactory moteurRenaultFact = new MoteurRenaultFactory();
MoteurFactory moteurCitroenFact = new MoteurCitroenFactory();

Voiture maRenault = new Voiture(moteurRenaultFact);
Voiture maCitroen = new Voiture(moteurCitroenFact);
```

Annexe 2

Creational patterns

◊ Abstract Factory / Factory Methods

Dans un langage fortement typé comme le C#, instancier un objet se fait par le mot clé `new`, suivi du type de l'objet. Admettons la classe Voiture qui instancie un moteur dès sa création.

Grâce au polymorphisme étudié plus tôt, nous avons vu qu'il était possible d'instancier une sous-classe tout en la considérant comme un objet de la classe mère. Cela implique que dans la suite logique du code, chaque objet qui instancie un objet de type Moteur doit connaître à l'avance quel type de sous-classe il doit utiliser.

Ne pourrions nous pas définir quelle sous-classe utiliser à un autre endroit du code ?

L'Abstract Factory nous aide ici à libérer la responsabilité de connaître explicitement dans la classe Voiture quel moteur elle doit utiliser. L'Abstract Factory est, comme son nom l'indique, une classe abstraite, qui propose des méthodes de création d'objet liés à son domaine (ici, les moteurs de voitures). On aura alors des sous-classes de cette factory pour chaque sous- classe d'objets instanciable.

La factory est passée en paramètre à la classe Voiture, ainsi la classe Voiture n'a plus à s'occuper de cas précis et peut accepter n'importe quel moteur pourvu qu'il corresponde à l'interface Moteur. Exemple du code source en Annexe 1.

La Factory Method correspond quasiment au même principe mais délégué uniquement sur une méthode. Cette méthode retournera souvent un objet différent en fonction de l'état de celui qui l'appelle.

◊ Singleton

Nous avons vu que dans la programmation orienté objet, les objets communiquent entre eux et se passent des messages. Chaque objet voulant traiter avec un autre doit avoir une référence vers celui-ci. Il existe cependant des cas où une information ou une action doît être partagée à l'ensemble du programme. Le Singleton répond à ce problème en proposant de rendre accessible un objet depuis n'importe où.

Le Singleton pattern s'assure d'instancier qu'une seule fois la classe ciblée et de partager cet objet lorsqu'il est demandé.

Miško Hevery (ingénieur chez Google), parle dans sa conférence Google Tech Talks du 13 novembre 2008 de la différence entre «Singleton» et «singleton».
Un Singleton est une instance contenue dans une variable statique et correspond donc à une variable globale. Le code est compilé de sorte qu'une seule instance soit possible.
Un singleton est une instance unique crée d'une classe, bien qu'il serait possible d'en instancier d'autre. Il est à la charge du développeur de s'assurer qu'une seule instance est crée, et non plus au compilateur.

Structural patterns

◊ Wrapper

Le Wrapper pattern (ou Adapter) consiste à étendre les possibilités d'un type d'objet non accessible (ou non modifiable) en l'encapsulant dans un nouveau type. Le Wrapper s'occupe de reproduire l'interface du type d'objet qu'il améliore, en déléguant les messages vers celui-ci, tout en ajoutant une nouvelle interface.

Cette solution permet aussi de faire du multi-héritage lorsque le langage utilisé ne permet pas de le faire, comme le C#. Le Wrapper reproduira alors les interfaces de chaque type d'objet qu'il contient.

◊ Facade

La Facade est un objet proposant une interface simplifiée d'un système complexe. Elle contient généralement plusieurs sous-systèmes indépendants cachés pour l'utilisateur. Ce pattern a pour but de faciliter l'accès à une opération complexe et de réduire les dépendances avec les objets des sous-systèmes.

Behaviour patterns

◊ Null object

Le Null object est un objet qui définit un comportement «neutre» par rapport à une interface précise.

Il arrive des cas où un objet n'a pas besoin d'être présent pour la continuité du code. Cependant, un appel vers un objet qui peut être présent ou non va générer une erreur (une NullReferenceException pour être précis) au runtime (c'est à dire lors de l'exécution du programme) lorsque l'objet en question sera défini comme `null`. Au lieu de définir l'objet cherché à `null`, on lui assigne un Null Object qui possédera l'interface désirée, mais dont son comportement ne fera rien.

◊ State / Strategy

Les patterns State et Strategy sont très ressemblants. Ces solutions permettent de permuter des comportements d'un objet au runtime tout en gardant la même logique.

Il y a deux éléments importants dans ce pattern : Les Comportements, et le Contexte.

Les Comportements sont contenus dans des objets répondant à une interface unique. Une classe abstraite est définie avec son interface (ses méthodes publiques) que chaque sous-classe implémentera à sa manière.

Le Contexte est l'objet de base qui utilise ces comportements. Cet objet possède une référence vers un objet du type mère du comportement (la classe abstraite). Grâce au polymorphisme, n'importe quel objet d'une des sous-classes peut être référencé et utilisé. Le Contexte appelle alors une des méthodes du Comportement définie par son interface. L'implémentation de cette méthode étant unique pour chaque sous-classe, on obtient alors une différence de comportement tout en gardant la même logique.

◊ Observer

L'Observer est un pattern proposant une solution de gestion d'événements. Un événement est exécuté lorsqu'une partie du programme réagit à un changement d'état dans une autre partie.

Il y a deux principaux acteurs dans ce pattern : L'observateur et l'observé. Ils seront déclarés comme des interfaces à implémenter pour chaque objet capable d'être l'un ou l'autre.

L'observateur s'enregistre auprès d'un observé dans l'attente d'une notification de celui-ci.

L'observé contient une liste d'observateurs qu'il notifie lors d'un changement d'état. Cette notification est gérée par le développeur.

Annexe 3

Dont be STUPID

STUPID représente l'acronyme de concepts à éviter lors du développement en POO.

Singleton : Nous avons vu plus haut que le Singleton était parfois considéré comme un anti-pattern, en voici un nouvel exemple.

Tight coupling : Le tight coupling (couplage serré) définit une trop forte dépendance entre les classes. Elle réduit les possibilités d'extension du programme ainsi que sa souplesse.

Untestability : Tester son code est aussi important que le développer. On procède alors à des «Unit Test» qui vont remplacer les données sensées être traitées par des données de tests. Rendre son code testable (et tester des parties spécifiques) devient une corvée lorsque celui-ci possède beaucoup trop de dépendance et donc de couplage serré.

Premature Optimization : Vouloir écrire du code optimisé dès le début rend souvent la lecture de celui-ci très difficile. Un code doit pouvoir être compris d'un développeur pour pouvoir être manipulé.

L'optimisation peut se faire dans des cas précis où la rapidité est nécessaire. Autrement, beaucoup de cas d'optimisations seront négligeables par rapport à la légèreté de l'écriture que proposerait une solution moins performante.

Indescriptive Naming : Écrire du code est similaire à écrire un livre. Si le lecteur ne peut pas déchiffrer un mot, le sens de la phrase lui semble difficile à cerner.

Prenons l'exemple d'un objet de type Computer, qui représente un ordinateur. Cet objet sera référencé par une variable pour son utilisation. N'est-il pas plus lisible de nommer cette variable `computer` que `aThing` ? ou même `cpter` ?

Duplication : La duplication du code entre en compte lorsque des objets ont besoin d'un algorithme ou d'une suite logique identique à un autre objet. Le problème est que si on décide de modifier cet algorithme, il faudra s'assurer de le modifier à tout les endroits où il aura été recopié. Préférez plutôt un refactoring de l'algorithme contenu dans un objet dont l'unique responsabilité est d'offrir le résultat de celui-ci.

SOLID

Single responsibility principle : Une classe devrait avoir qu'une seule et unique responsabilité. Si une classe possède plus d'une responsabilité, alors celle-ci peut être découpée en plusieurs autres classes.

Open/closed principle : Chaque entité d'un système devrait être ouvert à toute extension, mais fermée à toute modification.

Liskov substitution principle : Un objet dans un programme doît pouvoir être remplacé par un objet d'une de ses sous-classe sans altérer le bon fonctionnement de l'application.

Interface segregation principle : Proposer plusieurs interfaces spécifiques par application est préférable à une seule interface générale.

Dependency inversion principle : Un objet devrait déprendre d'une classe abstraite (définie par l'interface) et non d'une classe concrète (classe qui implémente l'interface par un code spécifique).

GRASP

GRASP est l'abréviation de «General Responsibility Assignment Software Patterns». Il définit des principes liés aux responsabilités qu'ont les classes dans un projet orienté-objet. Il se caractérise de différents patterns et concepts :

Controller : Un contrôleur doit pouvoir proposer les méthodes liées à un système dans le cadre du pattern Model-View-Controller (MVC).

Creator : Il est préférable de déléguer la création d'objet à une classe plutôt que d'instancier dans le code directement. Voir le Factory pattern.

Indirection : Pattern pour réduire le couplage serré en admettant un objet intermédiaire entre deux objets communiquant. Le «Controlleur» peut être une solution.

Information Expert : Utilise l'encapsulation pour cacher des informations d'un objet non nécessaire aux autres objets l'utilisant.

High Cohesion : La cohésion détermine le nombre de responsabilités pour une classe ainsi que sa possibilité de réutilisation, de maintenance et de compréhension. Une forte cohésion implique peu de responsabilité et donc plus de chance de modifier le programme sans réel impact.

Low Coupling : Implique qu'une classe doît dépendre au minimum à d'autres classes, dans le but de réduire les impacts de changements dans un programme, ainsi que d'augmenter la possibilité de réutilisation d'une classe dans un autre projet.

Polymorphism : Principe élémentaire de l'orienté-objet. Une classe définit des variations de comportements par l'utilisation de sous-classe en respectant l'interface de sa classe mère.

Protected Variations : Protège une classe des changements opérés dans un programme en utilisant des interfaces et/ou du polymorphisme.