

Rapport 2

Machi Koro



Sommaire

Introduction	2
1 UML - Seconde version	3
1.1 Ajouts	4
1.1.1 Packages	4
1.1.2 Classe Partie	4
1.1.3 Gestion de la pioche et du shop	4
1.1.4 Accesseurs	5
1.1.5 Classes énumération	5
1.2 Modifications	5
1.2.1 Gestion des pièces des joueurs	5
1.2.2 Gestion des cartes	5
1.2.3 Classe Edition de Jeu	6
1.2.4 Gestion des conteneurs de cartes	6
1.2.5 Arités des relations	8
1.3 Suppressions	8
1.3.1 Classe pièce	8
1.3.2 Package Effet	8
2 Travail de groupe	10
2.1 Cohésion de groupe	10
2.2 Nos réunions	10
2.3 Répartition des tâches	10
Conclusion	12

Introduction

Durant ce semestre et dans le cadre de l'UV LO21, nous allons travailler sur un projet de groupe. Notre projet de groupe répond au sujet disponible [ici](#). Le but de ce dernier est de nous familiariser avec la programmation orientée objet en produisant une version numérique du jeu *Machi Koro*. Nous mettrons en application dans ce projet les différentes notions vues en cours de LO21.

Dans ce second rapport, nous présenterons, dans un premier temps, nos analyses faites sur le sujet, à l'aide de notre second schéma UML. Ce schéma correspond à une amélioration de notre première version en prenant en compte nos points de vue quant à l'applicabilité de celui-ci.

Dans un second temps, nous expliquerons notre organisation au sein du groupe face aux différentes tâches. Vous pourrez, ainsi, retrouver notre liste des tâches mises à jour. Vous trouverez toutes nos versions du schéma UML en [annexe](#) du rapport.

est une responsabilité unique. Nos interfaces sont indépendantes, ce qui permettra de ne pas remettre en cause notre modèle en cas de nouvelle façon de jouer au jeu.

1.1 Ajouts

Afin de compléter notre UML avec les éléments manquants, nous avons procédé à certains ajouts, que nous avons réparti dans les différentes sous-sections.

1.1.1 Packages

Dans le but d'améliorer la lisibilité de notre UML, et de séparer notre jeu en plusieurs parties, nous avons opté pour la mise en place de packages. Le package **Contrôleur** est relatif à la gestion d'une partie. C'est la partie invisible pour les utilisateurs du jeu. Le package **Joueur** met en évidence l'ensemble des éléments relatifs au joueur.

Quant au package **Cartes**, il contient l'ensemble des éléments liés à la gestion des Cartes. En particulier, ce package comprend les packages **Monuments** et **Batiments**, ainsi que la classe carte. Le package **Monuments** contient la classe Monument et des exemples de classes, de même pour le package **Batiments**.

Nous avons également ajouté un package **Enum** contenant les différentes classes d'énumérations.

1.1.2 Classe Partie

Depuis la première version de l'UML, nous nous sommes interrogés sur la façon dont on devait modéliser une partie, et plus particulièrement l'entité qui allait la gérer. C'est pourquoi, nous avons créé une classe Partie. Elle sera chargée de veiller au bon déroulement de la partie, d'où sa présence dans le package Contrôleur.

Une partie aura pour attributs un tableau de joueurs, qui représentent les différents acteurs de la partie et un pointeur sur le joueur actuel. Par ailleurs, la classe aura un pointeur sur le joueur actuel, ce qui permettra d'identifier le joueur qui doit jouer. Ses méthodes lui permettront de jouer la partie, grâce à l'appel d'une fonction qui permet de jouer un tour et qui prendra en paramètre le joueur actuel. Enfin, il faudra établir une fonction permettant d'initialiser la partie. Cette fonction permettra de demander le nombre de joueurs, leur nom, d'initialiser les cartes du jeu et tous les autres éléments indispensables au fonctionnement de la partie.

1.1.3 Gestion de la pioche et du shop

Nous avons décidé de gérer la pioche ainsi que la réserve de cartes au centre de la table (que nous appellerons "Shop") permettant d'acheter des cartes. Le shop et la pioche composent tous deux la Partie. En effet, la classe Partie gérant l'ensemble du jeu, il est clair que la classe Partie est responsable du cycle de vie des objets Pioche et Shop. La pioche est définie avec un attribut contenu, qui est une pile de Batiment. La pioche disposera de méthodes lui permettant de s'initialiser, d'obtenir une carte, de savoir si la pioche est vide. Quant au shop, il aura un nombre de cartes possibles. La

représentation de l'attribut contenu sera explicitée ultérieurement dans le rapport. Le shop aura des accesseurs lui permettant d'interagir de manière optimale avec les autres classes comme la Pioche.

1.1.4 Accesseurs

Afin de pouvoir accéder aux attributs private depuis d'autres classes extérieures, il nous faut définir des accesseurs, en lecture (getter) et en écriture (setter). Ainsi, pour chaque classe de l'UML, nous avons défini les accesseurs lorsque jugé nécessaire.

1.1.5 Classes énumération

Afin d'améliorer la lisibilité de l'UML, nous avons opté pour la création de classes énumération. Ces dernières nous permettent de gérer les différentes valeurs de pièces possibles, les couleurs des bâtiments, le moment de l'effet ou le format du shop. Ces énumérations nous permettront de ne pas avoir à modifier tout le code en cas de changement de mode de jeu.

1.2 Modifications

1.2.1 Gestion des pièces des joueurs

Nous avons choisi de supprimer la classe pièce, car ses différentes caractéristiques pouvaient être intégrées à la classe joueur. Contrairement au fonctionnement dans la vraie vie, nous ne calculerons pas la somme d'argent d'un joueur à partir de ses pièces, mais plutôt l'inverse. Gérer un seul attribut "argent" de type entier sera plus simple et moins lourd que de gérer un ensemble d'instances de pièces. Nous aurons donc pour un joueur un attribut argent, et une méthode `get_repartition_pieces()` permettant d'obtenir dynamiquement la répartition des pièces d'un joueur à partir de son argent. L'affichage se fera aussi dynamiquement et en fonction de la répartition des pièces obtenue. Voici un exemple de ce que pourrait donner l'affichage des pièces pour un joueur ayant 28 crédits :



FIGURE 2 – Exemple d'affichage de pièces

1.2.2 Gestion des cartes

Dans la nouvelle version de l'UML, nous nous sommes rendus compte qu'il serait difficile de modéliser correctement chaque type d'effet avec des classes. Nous avons donc envisagé une modification de la classe **Carte** afin d'utiliser pleinement ses classes filles.

Ainsi, dans cette nouvelle version, une carte peut être modélisée par son image, son nom, son prix et la description de son effet. De ce fait, nous pouvons modéliser une carte ayant ses effets intégrés. Les effets seront déclenchés par la carte grâce à une méthode `declencher_effet()`. Les effets font donc dès à présent directement partie des cartes. Cette méthode sera déclarée dans la classe "Carte" comme une méthode virtuelle pure, qui sera ainsi redéfinie dans les différentes classes filles (Boulangerie, aéroport, etc). Étant donné qu'il n'y a plus de classe "Effet", cette dernière activera alors les différents effets sans passer par le biais d'une autre classe tierce. Nous trouvons que ce choix permettra une meilleure implémentation du jeu en utilisant la notion du polymorphisme d'héritage et en optimisant l'utilisation des classes filles afin de faciliter notre code.

1.2.3 Classe Edition de Jeu

Nous avons pu dans cette nouvelle version de l'UML étoffer et corriger notre classe Edition de Jeu. Ainsi, en plus des `getter` et du `setter menu()`, qui permet de récupérer les informations sur le mode de jeu d'une partie, nous avons ajouté de nouveaux attributs.

Tout d'abord, une liste de couples (Batiments,int) et (Monuments,int). Ces listes vont permettre de savoir le nombre et l'appartenance d'objets cartes déjà créés à une édition/extension de jeu. Ainsi, au lancement du jeu, toutes les cartes possibles seront créées dynamiquement, et c'est grâce à ces listes que seules certaines cartes seront sélectionnées en fonction du choix de l'édition de jeu par le joueur. L'attribut `est_edition` qui va permettre de faire la distinction entre une extension et une édition de jeu.

Ainsi, dans notre modélisation, nous considérons les extensions comme des éditions de jeu (car conceptuellement, elles fonctionnent de la même manière) seul l'attribut `est_edition` les différenciera. Enfin, nous avons aussi ajouté l'attribut `shop_format` qui va être un paramètre de la partie qui permettra de faire des variantes du jeu. Ce paramètre permettra de changer la manière dont le shop sera modélisé dans la partie.

Un shop est dit *extended* s'il n'y a pas de pioche et que toutes les cartes sont distribuées dans le shop au début de la partie et *standard* s'il est défini par un nombre de tas de cartes sur le plateau de jeu, les cartes restantes étant dans la pioche. Cet attribut permet aussi l'évolution du jeu. Si un futur programmeur souhaite implémenter une nouvelle variante du shop, il n'aura qu'à ajouter sa variante dans l'énumération `Shop_Format` puis modifier le constructeur de `Shop` à sa guise.

1.2.4 Gestion des conteneurs de cartes

En ce qui concerne les conteneurs de cartes, nous avons opté pour une solution qui minimise le temps de calcul et l'espace occupé en mémoire. Nous nous sommes inspirés du jeu de set vu en TD, et nous avons modélisé les cartes de façon à n'en créer qu'une seule instance de chaque carte.

Ces instances se trouvent dans les attributs bâtiment et monument de la classe `EditionDeJeu`, avec un attribut quantité pour gérer leur quantité. Ces attributs servent exclusivement à retrouver la définition et caractéristiques d'une carte donnée. Ailleurs dans le jeu, on n'utilise que des pointeurs vers ces cartes. Cela rend le jeu léger en mémoire car on ne crée jamais deux fois la même carte, créer 10 pointeurs vers une même carte prend beaucoup moins de place que 10 instances de cette carte.

Pour les mêmes raisons, le jeu est aussi plus léger en temps de calcul, le déplacement en mémoire d'un pointeur (ex : déplacement d'une carte du shop vers un joueur en cas d'achat) prend beaucoup moins de temps que le déplacement d'un objet entier. Pour rendre cela possible, il nous a cependant fallu retirer l'attribut `est_actif` de la classe `Monument`, pour le mettre dans un couple de valeurs dans la liste de Monuments du joueur. Sans cela, il aurait été impossible de ne créer qu'une seule instance de chaque monument, parce que le fait qu'un monument soit actif est propre au joueur qui l'utilise.

Pour bien comprendre les choix qui viennent d'être décrits, nous avons réalisé une figure :

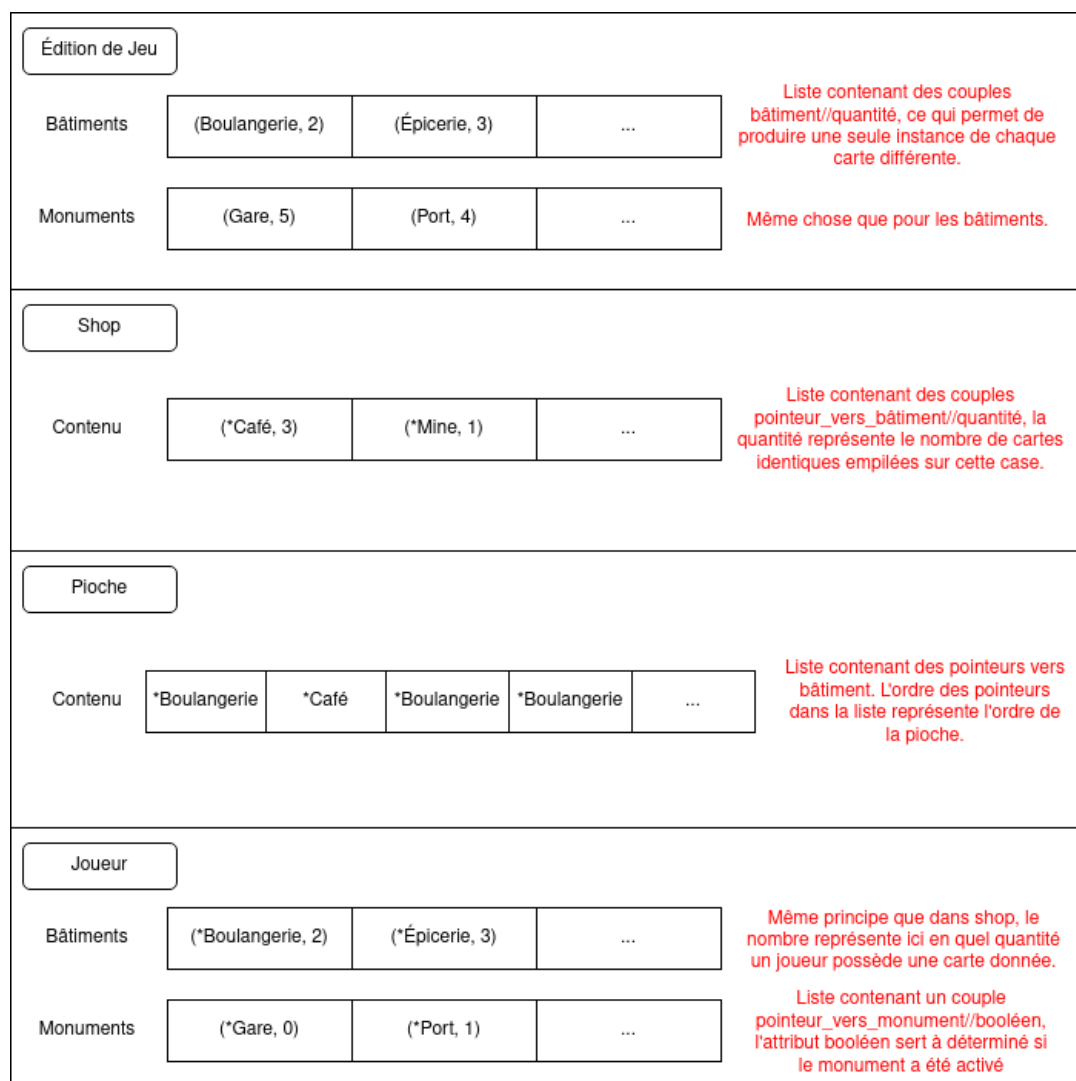


FIGURE 3 – Choix de modélisation pour les conteneurs de cartes

1.2.5 Arités des relations

Après réflexion par rapport à la première version de l'UML, nous avons effectué quelques modifications au niveau des arités de certaines relations, de manière à mieux correspondre à la réalité.

Pour commencer, nous avons modifié la relation entre Joueur et partie pour remplacer 1..* par 2..*. Nous conservons le nombre de joueurs maximum comme non fixe car celui-ci peut varier selon les éditions/extensions utilisées, mais comme le jeu se joue à deux joueurs au minimum, nous représentons cette règle dans la relation.

Nous avons également modifié le nombre de dés. La classe "Dé" n'est maintenant plus reliée à la classe "EditionDeJeu" mais plutôt à la classe "Partie", car c'est la classe partie qui gère les différents éléments permettant de jouer une partie. Le nombre de dés est passé de 1..2 à 1..*, car nous avons estimé que de futures extensions pourraient créer des cartes ayant des nombres d'activation nécessitant plus de dés.

Nous avons aussi dû gérer les arités des nouvelles classes "Shop" et "Pioche". Ainsi, une partie est composée de 0..1 pioche (il existe des versions du jeu où on distribue toutes les cartes dans le shop dès le début du jeu et où il n'y a donc pas de pioche). Une partie est composée de 1 shop et interagit avec. Une pioche (respectivement un shop) est constituée de * pointeurs vers des Bâtiments. Un pointeur de Bâtiment se trouve dans 0..1 pioche (respectivement shop).

Enfin, nous avons modifié l'arité entre Partie et EditionDeJeu. En effet, dans notre première version, nous considérons qu'une partie se joue avec une édition de jeu. Cela est vrai dans la réalité. Cependant, d'après notre modélisation où nous distinguons les extensions et les éditions, il est possible de jouer à la version Deluxe du jeu avec l'extension Green Valley. On joue alors 2 "EditionDeJeu". Ainsi, nous avons transformé la relation 0..1 en 1..*.

1.3 Suppressions

Certains éléments de notre précédent UML étaient erronés ou contradictoires avec notre future implémentation du jeu. Certains éléments ont donc été supprimés.

1.3.1 Classe pièce

Nous avons décidé de supprimer la classe pièce comme expliqué dans la partie Modification.

1.3.2 Package Effet

Nous avons aussi décidé de supprimer le package entier **Effet**.

En effet, comme énoncé dans le point 1.2.2, nous nous sommes rendus compte qu'il serait difficile de modéliser correctement chaque type d'effet avec des classes, et que cela pouvait être géré dans une méthode à l'intérieur de la classe Carte. De

plus, gérer les effets avec des classes nous compliquerait la tâche car chaque tour, nous devrions faire appel à la classe effet, puis à la classe spécifique de l'effet.

Puisque nous avons choisi de modéliser une classe par type de Carte, nous avons préféré intégrer l'effet à chaque classe définissant une Carte. Ainsi, la classe carte comporte une méthode virtuelle pure déclencher_effet() dont la définition sera faite dans les classes carte filles (Boulangerie, Foret, Mines...), selon les besoins de la carte en question. Même s'il y a un peu de redondance puisque deux cartes peuvent avoir des effets similaires, cela nous semble plus simple à implémenter et à utiliser.

Enfin, en insérant l'effet dans la définition d'une carte, l'ajout d'un nouvel effet (cas d'une nouvelle carte), par une nouvelle édition de jeu par exemple, est facilité.

Pour résumer, cette décision facilitera grandement la gestion du lien entre une carte et son effet, car celui-ci fera maintenant partie intégrante de la carte.

2 Travail de groupe

2.1 Cohésion de groupe

Dans un projet d'envergure comme celui-ci, il est nécessaire d'avoir une bonne cohésion au sein du groupe. Celle-ci nous permet de surmonter les différents problèmes pouvant survenir lors du projet. Nos différentes réunions permettent de veiller à ce que chacun puisse exprimer librement son avis et ainsi faire avancer ensemble notre vision du projet. Ces discussions nous permettent d'écouter les différentes personnalités du groupe, certains étant plus timides et d'autres plus extravertis.

Nos réunions permettent également de répartir équitablement le travail entre les différents membres, mais nous aborderons ce point dans la partie suivante.

Entre autres, notre cohésion nous permet d'éviter les tensions, de mieux se comprendre et de mieux communiquer. Cette communication facilite la compréhension des tâches que chacun souhaite effectuer pour avancer le plus efficacement possible.

2.2 Nos réunions

Pour ce projet, nous nous réunissons hebdomadairement. Dans ces réunions, nous discutons des différentes questions que nous nous posons, définissons et répartissons les tâches à faire pour la semaine à venir, et avançons en groupe sur le travail à réaliser.

Par exemple, nous avons convenu deux réunions pour concentrer nos efforts sur l'UML. La première, une session de 2 heures, nous a permis de faire émerger une vision commune du projet et notre implémentation. Puis avons donné en tâches pour le groupe que chacun de son côté trouvent des modifications à faire sur l'UML. Lors de la seconde réunion, nous avons mis en commun ce que nous avons trouvé pour produire l'UML final. Nous nous sommes aussi réunis afin de finaliser le rapport et se répartir les tâches suivantes du projet (commencer l'implémentation en code). En général, nous nous réunissons également après le TD dans le but discuter rapidement (entre 10 et 20 minutes) de l'avancement des parties de chacun sur le projet.

Ces réunions nous permettent d'avoir une vision claire sur les différentes tâches à réaliser et sur l'avancement de chacune d'entre elles. De plus, nous utilisons des outils collaboratifs, pour travailler ensemble efficacement et de centraliser nos différents travaux. Nous utilisons *Github* afin de centraliser nos fichiers et les nombreuses versions de ceux-ci ainsi que la plateforme libre développée par Picasoft, *CodiMd* pour l'ensemble de nos informations textuelles et schématiques. En effet, nous pouvons facilement retrouver sur ce fichier : notre to-do list mise à jour à chaque avancement, tous nos schémas UML et enfin nos notes contenant les points à aborder lors des réunions à venir.

2.3 Répartition des tâches

Dans cette partie, nous allons vous présenter la répartition des tâches depuis le début du projet. Pour le premier rapport, nous avons travaillé uniquement en groupe

car nous voulions que notre vision commune sur le projet soit claire. Ainsi, nous avons, comme dit plus haut, fait de nombreuses réunions à ce sujet. De ce fait, vous pouvez remarquer une nette différence entre le premier schéma UML ne contenant que le fruit de nos premières réunions et le second fait après de nombreuses réunions.

Pour ce rapport, nous avons commencé à réfléchir sur l'implémentation du jeu ensemble. Ainsi, nous avons pu commencer à développer les cartes bâtiments et monument. Nous nous sommes répartis comme suit :

- Nicolas : cartes bleues
- Antoine : cartes violettes
- Théo : cartes vertes
- Nasser : cartes rouges
- Sacha : cartes monuments

Conclusion

Au cours de cette deuxième partie du projet, nous avons réfléchi à la future implémentation du jeu Machi Koro. Plus particulièrement, cette phase a nécessité d'approfondir notre UML, afin de modéliser les éléments manquants et optimiser nos choix pour la suite.

Ce projet sera pour nous l'occasion de mobiliser les connaissances théoriques acquises lors des cours magistraux et travaux dirigés en les appliquant à un cas concret : design patterns (iterator, singleton), classes, méthodes, pointeurs et STL.

Le planning nous permettra de respecter les dates butoirs fixés par le projet. Quant à notre cohésion de groupe, elle nous permettra à nouveau d'être plus efficace dans le reste du projet. Ainsi, le gain de temps sera optimal lors de la troisième partie : création de l'architecture et du code.



Liste des tâches

Cette section est dédiée à l'avancée des tâches du projet. En particulier, on y trouvera les tâches à faire, celles en cours, et celles terminées. De semaine en semaine, nous reprendrons les différentes tâches et les feront changer de partie en fonction de leur état d'avancement.








Le format de chaque tâche est le suivant :

icone : tache ; personne en charge ; duree estime ; duree reelle






À faire :

-  : Correction de l'architecture si besoin ; tous ; 2 heures
-  : Créer l'interface graphique avec QtCreator et l'implémenter dans notre projet ; tous ; Durée Indéterminée

En cours :

-  : Modification de l'architecture avec l'intégration des extensions ; tous ; 3 heures
-  : Intégrer les premières classes en *C++* ; tous ; 6 heures
-  : Création des classes des cartes violettes ; Antoine ; 5 heures
-  : Création des classes des cartes rouges ; Nasser ; 4 heures
-  : Création des classes des cartes bleues ; Nicolas ; 4 heures
-  : Création des classes des cartes vertes ; Théo ; 5 heures
-  : Création des classes des cartes monuments ; Sacha ; 5 heures

Fait :

-  : Seconde version de l'UML ; tous ; 1 semaine
-  : Ajout des attributs et méthodes nécessaires au bon fonctionnement du jeu ; tous ; 3 heures
-  : Réalisation d'une esquisse d'UML ; tous ; 2 heures
-  : Répartition des tâches ; tous ; 30 minutes ; 1 heure
-  : Rédaction du rapport ; tous ; 4 heures

Annexe

Caractère	Icône pour le champ	Icône de la méthode	Visibilité
-	□	■	private
#	◇	◆	protected
~	△	▲	package private
+	○	●	public

FIGURE 4 – Visibilité des attributs et méthodes

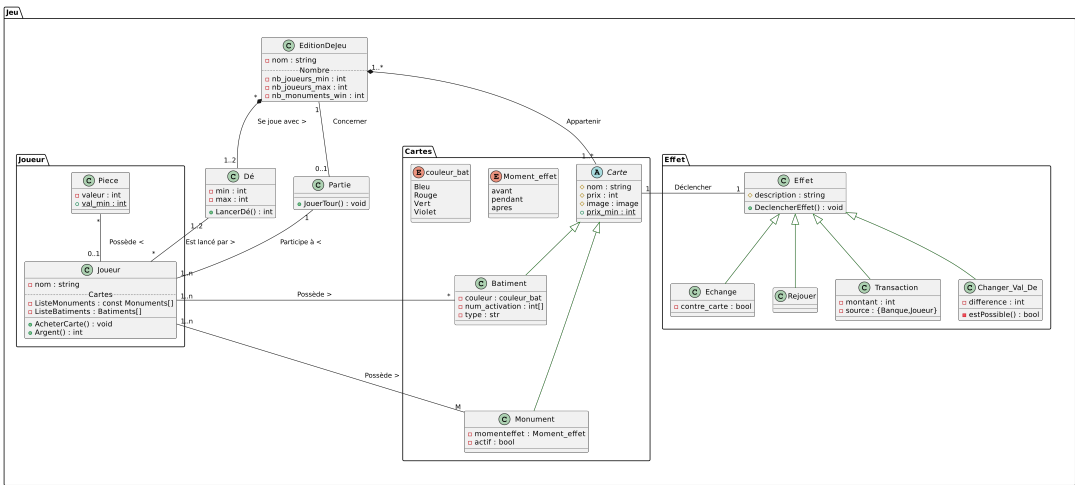


FIGURE 5 – Version définitive de l’UML 1

Table des figures

1	Diagramme UML d'une partie de <i>Machi Koro</i>	3
2	Exemple d'affichage de pièces	5
3	Choix de modélisation pour les conteneurs de cartes	7
4	Visibilité des attributs et méthodes	14
5	Version définitive de l'UML 1	14