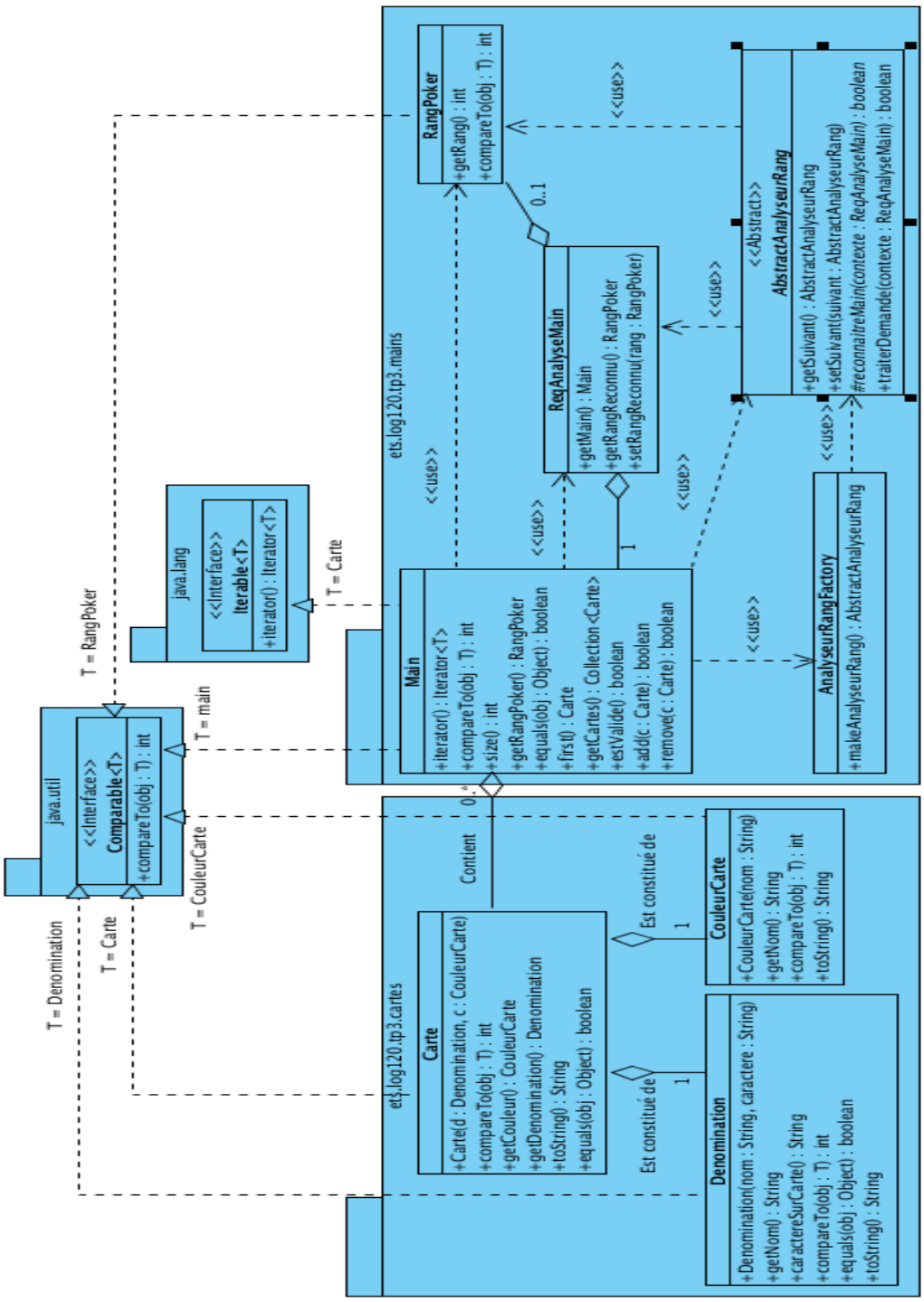
Université du Québec
École de technologie supérieure

Rapport de laboratoire

N° de laboratoire	3
Étudiant(s)	Samuel Milette-Lacombe et Martin Desharnais
Code(s) permanent(s)	MILS26059100 et DESM21099102
Cours	LOG120
Session	Automne 2011
Groupe	4
Professeur	Sébastien Adam
Chargés de laboratoire	Sébastien Déry, Carl-André Corbeil
Date	2011-11-21

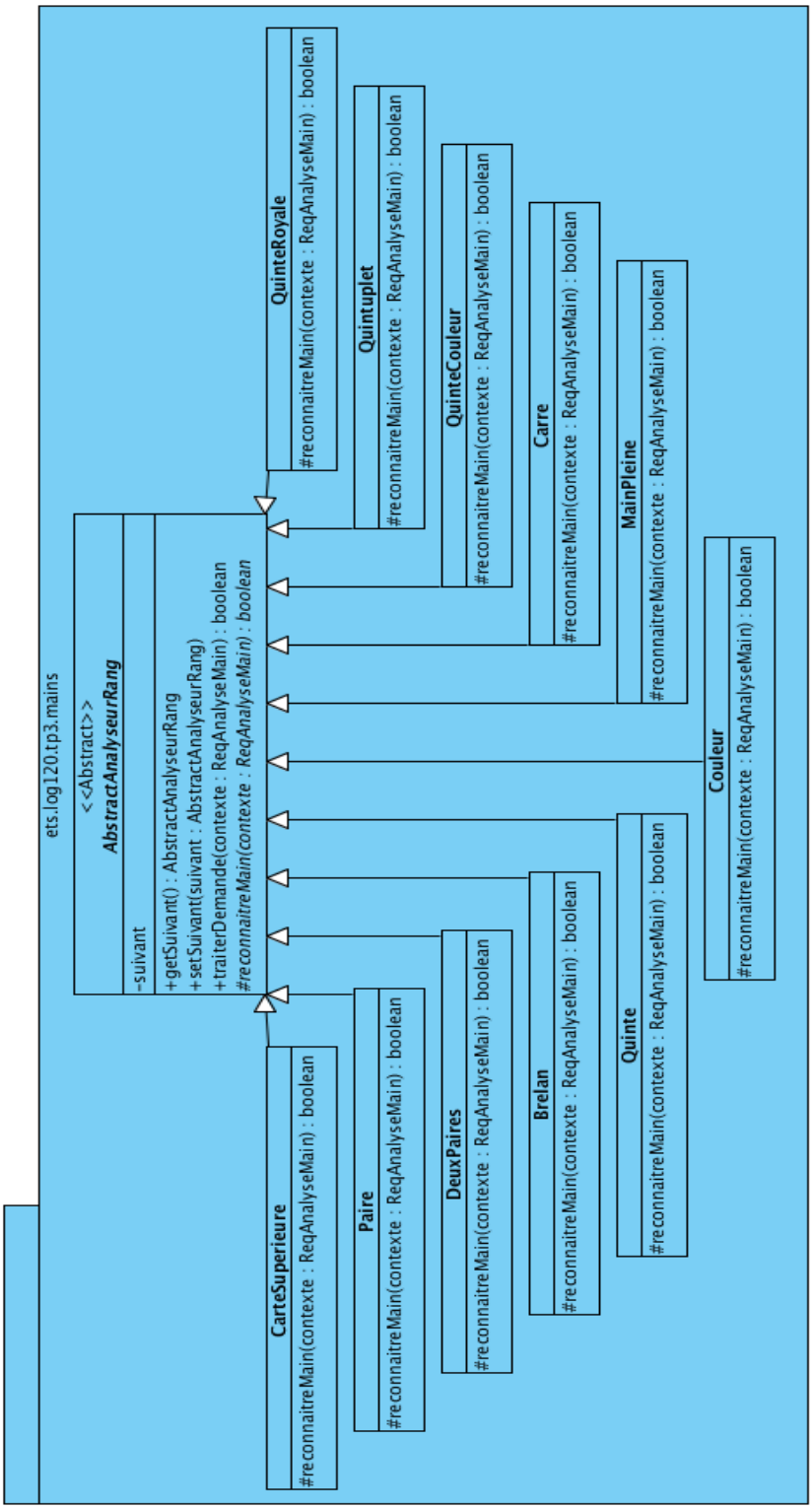
Diagramme(s) de classes

Diagramme de classes général



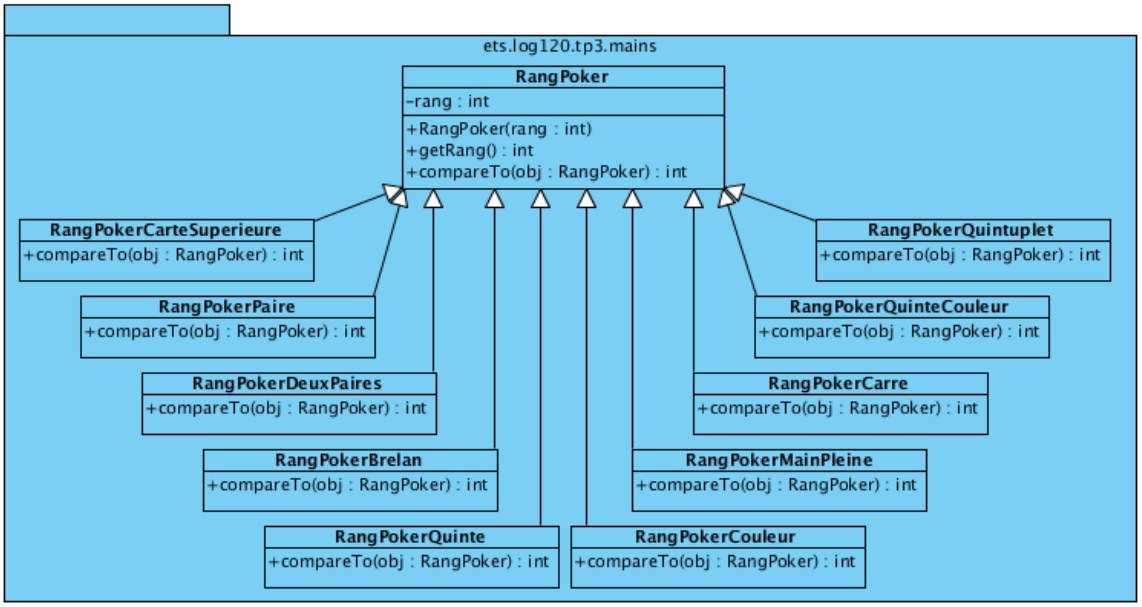
Le diagramme des classes général n'a pas énormément de différences par rapport à celui fourni avec la consigne du travail. La plus grande différence est que nous avons ajouté la classe « AnalyseurRangFactory » qui implémenter le patron « factory » pour créer la chaîne de responsabilités. Nous avons aussi défini différemment les relations entre les classes de ladite chaine : les classes « Main » et « AbstractAnalyseurRang » utilisent la classe « RangPoker » alors que celle-ci n'utilise personne, la classe « ReqAnalyseMain » agrège les classes « Main » et « RangPoker », etc. Nous avons aussi représenté les types interfaces « Comparable » et « Iterable » de la bibliothèque et avons représenté explicitement les classes qui les implémentent.

Digramme des classes représentant la chaîne de responsabilité



Le second diagramme représente les différentes héritant de « AbstractAnalyseurRang » et définissant la manière de détecter les différents rangs possibles sur une main. Dans les faits, c'est avec ces classes que la « factory » travaille pour créer la chaîne de responsabilités.

Diagramme des classes représentant les différents de poker.



Le troisième diagramme ne figurait pas du tout dans les spécifications initiales. Comme il sera expliqué plus loin, nous avons décidé d'implémenter une classe héritant de « RangPoker » pour la plupart des rangs possibles afin de pouvoir spécifier des moyens différents de les différencier.

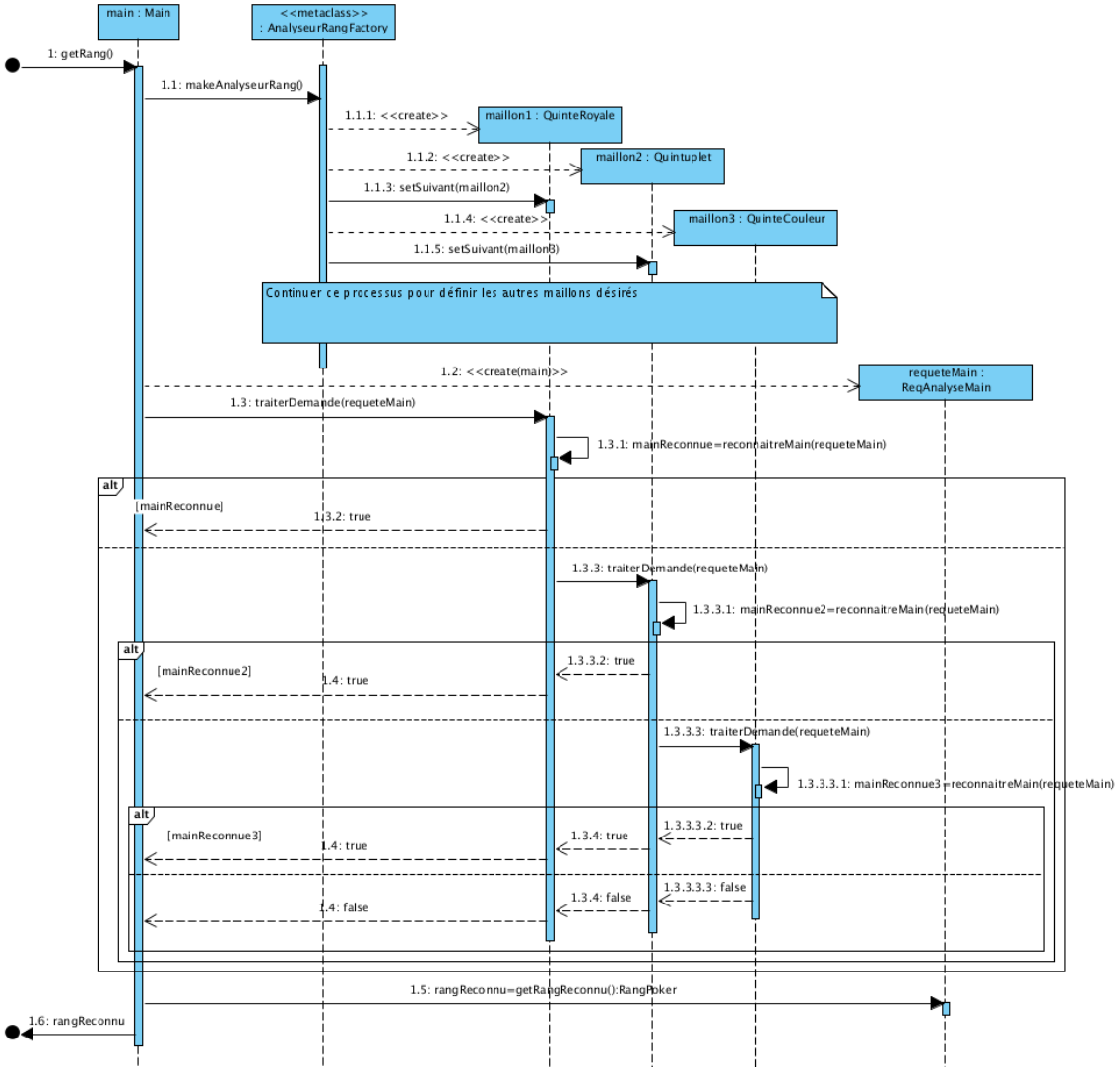


Diagramme de séquence :

Le diagramme de séquence représente l'exécution de la méthode getRang qui selon nous était le point de départ pour englober les évènements entrants dans la chaîne de responsabilité. En premier lieu, il y a un appel au "factory" AnalyseurRangFactory pour récupérer une instance utilisable de la classe AbstractAnalyseurRang. Ce factory créé des maillons prédéfinis représentants les handlers à parcourir identifiant des rangs à être présent dans les mains du jeu. Comme le diagramme l'indique avec une note, nous ne détaillons pas tous les maillons formant la chaîne de responsabilités.

Par la suite, pour chaque maillon, la méthode traiterDemande est exécutée. La méthode mainReconnue est ensuite exécutée et c'est le polymorphisme qui entre en jeu pour décider quelle méthode est utilisée. Si un rang est reconnu dans la main, il y a un retour de la valeur booléenne true à la main, si le rang à déterminer du maillon en cours

n'est pas reconnu dans la main il y a évaluation de la présence d'un maillon suivant et il se produit la même suite d'opérations évoquées précédemment. Dans le cas où aucun rang n'est reconnu et qu'aucun maillon suivant n'est présent (ne devrait pas arrivé), il y a un retour de la valeur booléenne "faux" à la main.

Décisions de conception

Décision 1 : Implémenter des classes enfants de RangPoker.

Nous devions mettre en place la comparaison des rangs de poker entre eux. Le plus simple des cas de figure était qu'un rang de poker soit comparé à un différent. Il s'agissait là uniquement de faire un tri sur le nombre entier représentant le rang. Cependant, en cas d'égalité sur le rang, il y avait d'autres critères qui entraient jeux comme la carte "kicker". La classe RangPoker à elle seule ne suffisait alors plus selon nous pour traiter la comparaison entre même rang.

Nous avons décidé de représenter chaque rang comme un enfant de la classe RangPoker. Ces classes dérivées peuvent contenir les attributs et la méthode compareTo personnalisée nécessaires à la bonne comparaison des rangs entre eux. Dans la méthode compareTo des classes dérivées, le rang défini dans RangPoker est évalué en premier pour déterminer s'il s'agit d'un même rang.

Il est à noter que les rangs comme Quinte Royale n'ayant pas de critères à évaluer en cas d'égalité, ne sont pas définis par une classe enfant puisque le numéro de rang suffit lors de la comparaison.

Au début, nous utilisions uniquement la classe RangPoker pour comparer les rangs entre eux. Cette classe contient uniquement un nombre qui identifie le rang. Nous nous sommes alors rendu compte qu'il fallait évaluer d'autres critères que le rang de poker identifié en cas d'égalité sur les rangs. Nous avions envisagé un instant à surcharger le constructeur de RangPoker pour pouvoir traiter plusieurs types de rang. Nous aurions eu par exemple un attribut pour chaque critère de comparaison entre rang (i.e. la carte kicker) qui aurait défini au moment de l'appel du constructeur désiré. Cependant, cette conception ne correspondait pas à nos idéaux de conception, nous nous sommes dit qu'une spécialisation de la classe RangPoker était plus appropriée. La spécialisation de rang poker avec le concept d'une classe par rang amène une bonne cohésion au programme. Les types de rangs sont sémantiquement séparés. Cette conception amène aussi à une modularité supplémentaire du programme. Qui dit modularité dit généralement une meilleure possibilité d'extensibilité du logiciel. Nous en reviendrons dans la partie du l'extensibilité de notre logiciel.

Décision 2: Utilisation du patron factory pour la chaîne de responsabilité.

La chaîne de responsabilité est définie par la classe abstraite AbstractAnalyseurRang et de ses "handlers" définies par des rangs de poker. Pour pouvoir utiliser l'analyseur dans la classe main, il nous fallait pouvoir utiliser un objet non abstrait pour pouvoir travailler avec identifier le rang des mains. Nous avons choisi d'utiliser le patron "factory" défini dans la classe "AnalyseurRangFactory" afin de pouvoir travailler sur un objet prêt à l'utilisation afin d'exécuter la méthode traiterDemande qui permet d'identifier le rang. Cette manière de faire permet de hausser le niveau d'abstraction de l'utilisation de la chaine de responsabilité. L'utilisateur du cadriciel n'a qu'à personnaliser sa suite de rang à analyser dans la classe AnalyseurRangFactory. L'objet "AbstractAnalyseurRang" est alors tout le temps instancié de la manière qu'on le désire peut être utilisé dès son appel. À moins qu'il le veule vraiment, l'utilisateur du cadriciel n'a donc pas à gérer en cours d'exécution les maillons de la chaîne pour l'analyse des rangs.

Décision 3: Utiliser des énumérations pour les dénominations et couleurs de carte.

Les spécifications fournies avec l'énoncé de travail demandaient d'implémenter les dénominations et couleurs de carte en tant que classes. De plus, les exemples de code fournis nous ont appris que les constructeurs de celles-ci devraient être privés et qu’un certain nombre d'instances devraient être publiques sous forme de constantes.

Selon nous, les concepts de dénomination et de couleurs de carte auraient pu être mieux représentés par des énumérations. Celles-ci fonctionnent, en java, selon le même principe qu'une classe : elles peuvent avoir des méthodes, des attributs, etc. Seulement, contrairement aux classes, elles fournissent une vérification à la compilation et peuvent être utilisées dans des structures conditionnelles « switch ». De plus, la méthode « toString » est déjà implémentée par défaut pour retourner le nom de la constante.

Décision 4: Comparer deux cartes entre elles

Les spécifications fournies avec l'énoncé de travail indiquaient que les classes « Denomination », « CouleurCarte » et « Carte » devaient toutes trois fournir la méthode « compareTo » et, par conséquent, implémenter l'interface « Comparable ». Seulement, ce n'est pas, selon nous, leur responsabilité de

pouvoir se comparer entre elles. En effet, un deux de carreau peut ne pas valoirgrand chose dans un jeu de cartes, mais en valoir beaucoup dans un autre.

Le code fourni en exemple utilisait une liste interne pour savoir quelles dénomination, ou couleur, valaient plus qu'une autre. La première solution pour permettre à l'utilisateur serait de lui exposer cette liste pour qu'il puisse en modifier l'ordre.

Seulement, nous croyons qu'une dénomination et une couleur de carte ne devraient même pas pouvoir être comparées. Pour être plus précis, cette tâche devrait être laissée à l'utilisateur de la bibliothèque qui décidera, selon le contexte (selon le jeu) quel critère de comparaison utiliser. Pour ce faire, deux solutions sont possibles : déplacer ce code dans une autre classe ou permettre à l'utilisateur de personnaliser les cartes, dénominations et couleurs de cartes.

La première solution pourrait être mise en place en délégant cette responsabilité à la classe « main ». En effet, comme il est probable que deux jeux de cartes nécessitent des classes main différentes, celles-ci ont toutes les informations nécessaires pour comparer les différents éléments. Le problème de cette solution est qu'elle apporte un fort couplage entre les cartes et la main : celles-ci ne peuvent presque plus être utilisées séparément.

La seconde solution semble plus cohésive, en plus de diminuer le couplage. Il suffit de demander à l'utilisateur comment il souhaite comparer ses cartes. La solution la plus évidente serait de demander un ou des comparateurs dans le constructeur de la classe « Carte », mais cette solution s'avère très verbeuse et peut pratique pour l'utilisateur qui doit toujours fournir les mêmes informations. Il serait aussi possible d'offrir d'exposer un attribut statique qui pourrait n'être défini qu'une seule fois. Seulement, cette solution rend totalement impossible d'utiliser simultanément deux règles de comparaison.

La meilleure solution serait donc de créer une classe « BasicCarte » qui utiliserait des « generics » pour laisser à l'utilisateur la possibilité de définir ses comparateurs. De plus, pour éviter que celui-ci n'ait à le répéter à chaque instanciation, il luit suffirait de définir un « typedef ». Malheureusement, cette notion n'existant pas en java, il faudrait utiliser un peu de ruse en définissant simplement une sous-classe héritant de « BasicCarte » et définissant les types des comparateurs.

Implémenter cette conception s'est avéré assez difficile compte tenu du manque d'expérience avec les « generics » java. De plus amples recherches seraient nécessaires pour savoir ce que l'on peut faire et ce que l'on ne peut

pas avec ceux-ci. Pour pouver que le concept est valide, nous l'avons implémenté avec des « templates » en C++ (c.f. comparerDeuxCartes.cpp).

Décisions d'implémentation

Décision 1: Utilisation d'une fonction pour compter le nombre de dénominations

Durant le développement du cadriciel de poker, nous avions à programmer des rangs de poker revenant souvent à compter le nombre de chaque dénomination dans une main. Nous nous sommes donc retrouvés rapidement avec une répétition de code qui faisait la même chose. C'est pourquoi il nous est venu à l'idée d'unifier tout ce code faisant à peu près la même chose, dans une seule méthode placée dans une classe définissant des méthodes utiles pour l'analyse de main. Cette méthode se nomme "compterDenominations" et se trouve dans la classe "AnalyseurUtil". La méthode prend en paramètre une main et retourne une map de type "TreeMap". Cette classe a une implémentation toute particulière, elle utilise une "map" pour stocker sous forme de clé valeur, la dénomination et son nombre d'occurrences dans la main. L'implémentation consiste en un parcours des cartes de la main avec une boucle "for each" qui va remplir la "map" en incrémentant les occurrences des dénominations rencontrées. L'utilisation de cette méthode n'est pas seulement limitée pour les rangs devant avoir tant de dénominations. En effet, cette méthode sert aussi à compter le nombre de joker, une information essentielle pour beaucoup de nos classes de rang poker pour pouvoir gérer ces frimes.

Voici le pseudo-code correspondant à notre fonction de comptage de dénomination utilisant une "map".

```
FONCTION compterDenominations RECOIT une main
        INITIALISER une map d'ordre décroissante avec dénomination comme clé et un nombre entier pour valeur

        POUR CHAQUE carte DANS main
                obtenir le nombre de dénomination de la carte
                ajouter le nombre de dénominations à la map à la clé "denomination de la carte"
        FIN POUR CHAQUE

        RETOUR map
FIN FONCTION
```

Décision 2: Utiliser un multiset pour stocker les cartes dans la main

Lorsqu'est venu le temps d'implémenter la main, nous avons eu à choisir sous quelles formes les stockers. Avant de faire un choix, nous avons listé les contraintes que notre solution allait devoir satisfaire : contenir un nombre arbitraire de cartes,

permettre de les parcourir à l'aide d'un itérateur, permettre de les trier et permettre les doublons.

Les premiers et seconds critères éliminent automatiquement l'idée de prendre un simple tableau. Dès lors, il fallait choisir entre les différents conteneurs mis à notre disposition. La première idée émise fut d'utiliser un « ArrayList ». Celui-ci respecte toutes les contraintes proposées, mais nécessite d'être trié manuellement à chaque ajout ou suppression de données.

Nous nous sommes alors penchés sur la classe « TreeSet » qui respecte presque toutes les contraintes fixées. La seule exception est la nécessité de permettre les doublons. Cependant, cette contrainte est apparue assez tardivement dans le développement, lorsque nous avons voulu stocker plusieurs jokers dans un main. Avant cela, nous souhaitions que chaque carte soit unique, ce nous aurions dû faire manuellement avec un « ArrayList » mais que le « TreeSet » assure automatique. De plus, ce dernier a pour avantage d'aussi assurer automatiquement que les éléments sont triés.

Tous ces avantages nous on fait choisir d'utiliser un « TreeSet » pour stocker les cartes pendant la majorité du projet.

Cependant, comme dit précédemment, nous sommes revenus sur nos critères initiaux lorsque nous avons ajouté la gestion des jokers et avons décidé qu'il fallait permettre que la même carte se retrouve deux fois dans une main. Cette décision permet aussi de gérer des éventuelles variantes du jeu de poker qui nécessiteraient plus d'un jeu de cartes.

Ayant été très satisfaits du « TreeSet » jusqu'alors, nous avons décidé d'utiliser une de ses variantes, le « TreeMultiSet », dont la seule différence est de permettre le dédoublement des données.

Nous avons cependant vite déchanté lorsque, après de nombreuses recherches, nous avons découvert que la bibliothèque standard Java ne fournit pas ce conteneur. Nous avons donc décidé d'utiliser la bibliothèque « Guava » de Google qui vient fournir, en autres choses, ce conteneur.

Décision 3: Écrire une fonction générique pour détecter les quintes

Parmi les différents rangs des mains de poker qu'il fallait pouvoir détecter, trois étaient fortement liés : quinte, quinte couleur et quinte royale. La quinte est définie comme une séquence de cinq quartes, la quinte couleur est une quinte dont toutes les cartes sont de la même couleur et la quinte royale est une quinte couleur dont la carte la plus haute est l'as.

Les classes chargées de détecter ces trois rangs devaient donc détecter la même chose, en ajoutant simplement un critère supplémentaire chacun. Ne souhaitant pas dupliquer le code de base nécessaire pour détecter une quinte, nous avons donc cherché à l'extraire dans une fonction, que nous appellerons « trouverQuinte », qui pourrait être utilisée par les trois classes.

Parmi les possibilités que nous avons explorées, nous avons envisagé que la fonction prenne une main en paramètre et retourne la dénomination de la meilleure carte de la quinte si elle en trouve une et null dans le cas contraire. Il aurait ainsi été facile de détecter une quinte royale, mais totalement impossible de détecter une quinte couleur.

Nous avons aussi envisagé que la fonction retourne un « ArrayList » contenant les cartes de la meilleure quinte. Il serait alors possible de vérifier si la meilleure carte est un as et de vérifier si toutes les cartes sont de même couleur. Le problème est que si une variante du poker où le nombre de cartes par main est plus important, il devient possible qu'il y ait deux quintes dans une main. La fonction « trouverQuinte » ne retournant que la meilleure quinte, il est alors impossible de savoir, dans le cas où la première quinte n'est pas une quinte couleur, si la deuxième pourrait fonctionner.

Il pourrait aussi être envisagé de retourne un « ArrayList » contenant toutes les « ArrayList » de quintes trouvées. Cette solution n'est pas très attirante, car elle nécessite de toujours parcourir toutes les cartes et d'instancier de nombreux conteneurs.

La solution la plus élégante que nous avons trouvée est de fournir un itérateur, représentant le début de la séquence de cartes à analyser, à la fonction « trouverQuinte ». Celle-ci peut alors chercher une quinte dans cette séquence et, dès qu'elle en trouve une, retourne un itérateur représentant le début de la quinte. Dans le cas où la séquence de cartes ne contient aucune quinte, la fonction « trouverQuinte » retourne simplement « null ».

Le code appelant peut alors vérifier si l'itérateur retourné est « null » pour savoir s’il y a une quinte, vérifier si la première carte de la quinte est un as pour savoir si c'est une quinte royale et parcourir la quinte pour savoir si toutes les cartes sont de même dénomination et, donc, s'il s'agit d'une quinte couleur. Jusque-là, il n'y a aucun avantage par rapport à la fonction retournant un « ArrayList », mais c'est sans compter le cas où nous cherchons une quinte couleur ou une quinte royale et que celle-ci n'est pas la première de la main.

Puisque nous avons un itérateur pointant sur la première quinte et que nous avons dû la traverser pour savoir si toutes les cartes sont de la même couleur, nous pouvons rappeler la fonction « trouverQuinte » en lui passant ce même itérateur : celui-ci pointant après la quinte. La seconde analyse ne parcourra pas toute la main, mais seulement la section qui n'avait pas encore été analysée, l'analyse s'étant terminée dès qu'une quinte a été trouvée.

Pour rendre le tout un peu plus concret, voici le pseudo-code de la fonction « trouverQuinte » :

```
FONCTION trouverQuinte RECOIT l'itérateur du début de la séquence
        INITIALISER un conteneur vide représentant la quinte potentielle
        TANT QUE nous n'avons pas atteint la fin ET qu'aucune quinte n'a été trouvée
                SI la quinte est vide
                        ajouter la carte à la quinte
                        sauvegarder l'itérateur du début de la quinte
                SINON SI la carte courante suit la dernière carte de la quinte
```

```
                ajouter la carte à la quinte
            SINON
                vider la quinte
                ajouter la carte à la quinte
                sauvegarder l'itérateur du début de la quinte
            FIN SI
            déplacer l'itérateur vers la carte suivante
        FIN TANT QUE

        SI nous avons trouvé une quinte
            RETOUR de l'itérateur pointant sur le début de la quinte
        SINON
            RETOUR d'un itérateur nul
        FIN SI
FIN FONCTION
```

Cependant, nous avons éprouvé des problèmes au moment d'implémenter cette fonction en java : la copie de l'itérateur qui doit être faite pour retourner le début de la quinte ne l'était pas. En réalité, nous obtenions deux variables pointant sur le même itérateur : déplacer le premier déplacé aussi le second. Il était donc impossible de retourner un itérateur pointant sur la première carte de la quinte. Nous avons fait plusieurs recherches ,fait de nombreuses tentatives et nous sommes rendu à l'évidence que les itérateurs java n'ont pas été conçus pour ce genre d'utilisation. Des recherches nous ont appris que si les concepteurs de la bibliothèque standard Java ne nous permettent pas d'effectuer des copies d'itérateur, c'est parce qu'il sont incapables d'effectuer cette copie de manière performante.

Nous avons envisagé d'écrire notre propre itérateur copiable, mais nous avons le même problème qu'eux : ce n'est vraiment pas performant. Le temps filant, nous avons laissé ce problème sur la glace et avons pris la méthode simple, mais imparfaite de simplement copier le code commun d'une classe à l'autre.

N'ayant pas pu faire fonctionner la fonction, il nous était tout de même impossible d'accuser ouvertement Java. En effet, peut-être que c'est nous qui avions fait une erreur. Pour savoir si cette idée aurait dû fonctionner, j'ai donc entrepris de l'implémenter dans un autre langage et, en moins de dix minutes, la fonction était fonctionnelle. Le concept est donc bon, mais n'est pas dans la « philosophie » java.

PS: le code fonctionnel de la fonction et deux exemples d'utilisation se trouve à la racine du projet sous le nom « trouverMeilleureQuinte.cpp ».

Discussions

Les patrons GOF

La bibliothèque de jeux de poker que nous avons mis au point à été conçue de manière à ne gérer aucune règle spécifique. Des classes pour représenter les différentes cartes, les agencer en mains, et comparer deux mains entre elles sont fournies, mais ne savent pas comment comparer les mains entre elles. C'est à l'utilisateur de la bibliothèque de définir le comportement à adopter en fonction des différentes main.

Celui-ci est en effet invité à définir des classes, héritant de « AbstractAnalyseurRang », représentant les différents rangs qu'il souhaite gérer. Dans ces classes, il est de sa responsabilité de détecter si la main contient bel et bien le rang cherché et, si oui, à en informer la bibliothèque en insérant une instance de la classe « RangPoker » dans le contexte de recherche. Les « RangPoker » seront alors utilisés pour distinguer les rangs entre eux.

Advenant le cas où l'utilisateur souhaite une détection plus fine, il peut définir des classes spécialisant « RangPoker » pour pouvoir départager deux rangs pourtant égaux. Par exemple, si une main contient une paire de quatre alors que l'autre contient une paire de cinq, la classe « RangPoker » les considérera comme égales puisque les deux mains contiennent une paire. Au contraire, une hypothétique classe « RangPokerPaire » pourrait, en plus comparer la dénomination des deux paires pour les départager.

Il est donc suggéré d'écrire des classes de rang, héritant de « RangPoker » et d'analyseur, héritant de « AbstractAnalyseurRang » ensemble afin de pouvoir départager le plus précisément possible les deux cas.

Ces classes, une fois implémentées, doivent être agencées afin de pouvoir former une chaîne de responsabilité. Actuellement, la classe « Main » utilise la classe « AnalyseurRangFactory », qui implémente la méthode « makeAnalyseurRang », pour obtenir la chaine de responsabilité. Il serait judicieux, dans le futur, de modifier cette situation afin que la classe « AnalyseurRangFactory » soit un type interface et que ce soit l'utilisateur qui en fournisse un exemplaire à la classe « Main ».

Ce qu'il faudrait changer pour supporter les frimes.

Notre compréhension de l'énoncé du laboratoire était que nous devions gérer les frimes. Nous allons donc expliquer comment nous avons fait pou les traiter.

Conceptuellement, nous avons représenté la frime comme étant une carte composée d'une dénomination « joker » et d'une couleur « joker ». Une carte identifiée comme telle peut facilement être détectée dans les algorithmes de comparaison des rangs. Étant donné que nous utilisons un conteneur « multiset » pour stocker nos cartes, la carte joker allait devoir elle aussi se faire trier. Elle représente donc la plus haute carte du jeu bien qu'elle n'ait pas de valeur définie puisqu'il s'agit de la carte la plus pratique du jeu.

Initialement, nous ne prenions pas en compte les frimes, nous les avons prises en charge seulement en dernier. Il a été nécessaire de modifier une partie non négligeable de notre code pour les gérer. Tous nos algorithmes de comparaisons des rangs doivent compter le nombre de jokers présents dans la main. Pour cela, nous avons souvent utilisé la fonction « compterNombreDenominations » qui nous fournit, entre autres, cette information. En sachant le nombre de jokers dans la main, un traitement est effectué pour prendre en compte que certaines suites ou combinaisons peuvent être complétées avec l'ajout d’une ou plusieurs frimes.

La prise en charge des frimes dans les classes de notre bibliothèque a quelques points qui pourraient être améliorés. Le premier est minime, puisqu'il aurait fallu quelques lignes de code pour régler le problème. Le constructeur de la classe « Carte » prend en paramètre une dénomination et une couleur. Cependant, ce constructeur ne valide pas les dénominations et les couleurs passées en argument. Cela met en jeu la cohérence des cartes. En effet, on pourrait passer au constructeur une dénomination « joker » et une couleur autre que plaisanter. Il s'agit même de ce qui était arrivé dans nos tests unitaires, l'utilisation du copier-coller de cas tests où l'on a remplacé que la dénomination ordinale de la carte par celle du joker nous a permis de mettre en lumière ce problème. Il aurait fallu interdire les dénominations et les couleurs « joker » dans le constructeur et obliger, pour obtenir un joker, d'utiliser la constante statique définie à cet effet.

Ce qu'il faudrait pour prendre en charge les mains à sept cartes.

Nous avions compris que nous ne devions pas gérer un nombre fixe de cartes. Cela nous a donc menés indirectement à orienter notre conception et notre implémentation afin de prendre en charge les mains à sept cartes. Il s'agissait d'une difficulté supplémentaire et nos algorithmes on dû être plus complexes. En effet, pour les quintes par exemple, établir quelle est la quinte

est relativement facile, mais établir quelle est la meilleure quinte l'est moins puisque plusieurs quintes sont possibles dans une main de plus de 9 cartes.