

Sommaire

- **I - Interfaces Homme-Machines**

- 1.1 Contexte
- 1.2 Personas & user story
- 1.3 Diagramme des Cas d'utilisation
- 1.4 Analyse du diagramme de cas d'utilisation

- **II – Conception Orientée Objet**

- 2.1 Diagramme de paquetage
- 2.2 Analyse du diagramme de paquetage
- 2.3 Diagramme de classes
- 2.4 Analyse du diagramme de classes
 - 2.4.1 Le cœur de l'application : les astres
 - 2.4.2 Les fabriques d'astres
 - 2.4.3 Les constellations
 - 2.4.4 Le Manager et la Carte
 - 2.4.5 Les classes utilitaires
- 2.5 Diagrammes de séquences
 - 2.5.1 Relier deux étoiles (vue d'ensemble)
 - 2.5.2 Relier deux étoiles
 - 2.5.3 Supprimer une étoile dans une constellation
 - 2.5.4 Supprimer un Astre
- 2.6 Analyse des diagrammes de séquences
 - 2.6.1 Relier deux étoiles (vue d'ensemble)
 - 2.6.2 Relier deux étoiles
 - 2.6.3 Supprimer une étoile dans une constellation
 - 2.6.4 Supprimer un Astre
- 2.7 Description écrite de l'architecture
 - 2.7.1 Architecture globale du programme
 - 2.7.2 Dépendances
 - 2.7.3 Patrons de conception

- **III – Projet Tuteuré**

- 3.1 Diagramme de paquetage sur la persistance
- 3.2 Analyse du diagramme de paquetage
- 3.3 Diagramme de classes sur la persistance
- 3.4 Analyse du diagramme de classes
 - 3.4.1 Le Stub
 - 3.4.2 Aparté sur le patron de conception
 - 3.4.3 Persistance en DataContract To XML
 - 3.4.4 Persistance en JSON
- 3.5 Diagramme de classes sur les parties ajoutées
- 3.6 Vidéo promotionnelle

I - Interfaces Homme-Machines

1.1 Contexte :

Nous cherchons à concevoir une application nommée *Stellar*. Elle porterait sur le thème de l'astronomie.

Nous avons conscience qu'il existe aujourd'hui de nombreuses applications sur l'astronomie, et c'est pourquoi nous avons voulu faire une application originale, utile et pratique, qui offre deux possibilités. Cette application se décompose donc en deux parties : une partie à objectif informatif et une partie à objectif créatif sous forme d'un éditeur.

Tout d'abord, on aurait une carte du ciel sur laquelle serait placé des étoiles et planètes, ou plus vaguement des *astres*. On aurait la possibilité d'obtenir différentes informations sur une étoile spécifique, telle que sa température, sa masse, son nom, sa constellation (si elle en a une), sa date de découverte, de même pour les planètes avec sa galaxie, etc.

L'application a également été pensée ergonomie, c'est pourquoi il serait possible de rechercher un astre en fonction de divers filtres (dont un système de favoris qui permet de sauvegarder les astres préférés de l'utilisateur).

La seconde partie consiste à ce que l'utilisateur puisse créer les étoiles et planètes qu'il souhaite (qu'elles existent ou non, il est libre de faire ce qu'il souhaite). Il peut ainsi les placer sur la carte, relier des étoiles entre elles pour créer des constellations, les supprimer... Il pourra aussi enregistrer son travail, puis le charger et le reprendre plus tard. Bien sûr, il lui sera possible d'exporter sa carte en image, afin qu'il puisse s'en resservir dans un projet s'il le souhaite, il peut aussi imprimer sa carte créée afin de pouvoir l'afficher chez lui !


Cette application permet à n'importe quel utilisateur s'intéressant à l'astronomie (de façon amateur essentiellement) de pouvoir obtenir des informations sur les étoiles.

Les développeurs qui ont des besoins spécifiques lors de création de jeux, d'univers, peuvent utiliser le côté éditeur de l'application afin de se servir du ciel créé dans des projets. Le caractère initial de l'éditeur peut également être contourné pour être utilisé comme logiciel de dessin de formes géométriques (on peut imaginer les étoiles comme des points, et les traits comme des segments).

Les personnes s'intéressant à l'astronomie et souhaitant apprendre sur les étoiles et planètes pourront utiliser le côté informatif de l'application pour s'instruire sur les astres qui peuplent notre galaxie, notamment lors de balades nocturnes car l'application ne nécessite aucune connexion internet.

Les écoliers (primaire) sont les cibles typiques de cette application, car elle se présente sous forme ludique et laisse libre cours à l'imagination. C'est donc une bonne application pour découvrir l'astronomie et apprendre en s'amusant, en utilisant toutes les fonctionnalités mises à disposition !

1.2 Personas & user story



Vincent Time

Je me renouvelle au contact de la nature et des gens

Age : 33 ans

Profession : sans emploi

Revenus : RSA

Situation : marié et père de deux enfants.

Ancien fonctionnaire, Vincent suit une formation de paysagiste pour se réorienter. Il adore les randonnées, treks et trails d'endurance, et son contact avec la nature est une source de vitalité, c'est aussi pour cela qu'il souhaite changer de métier.

Internet : fréquence basse d'utilisation

Sites préférés :

- photographesdumonde.com
- tela-botanica.org

Applications préférées :

- Ecobalade
- Stellarium

Passe-temps :

- Voyager, découvrir des paysages, des personnes, des cultures...
- Apprendre à connaître le monde qui l'entoure (plantes, animaux)
- Balades nocturnes avec sa famille

Profil technique :

- Assez bonne maîtrise des outils bureautiques (suite Microsoft).
- Aucune notion de programmation.
- Possède un téléphone Android Huawei P9, une tablette Samsung et un ordinateur sous Windows 10.
- Passe environ 1h/j sur les écrans.
- Se forme en continu sur les métiers d'extérieur.
- Aime partager ses connaissances avec sa famille lors de randonnées.

C'est lors d'une randonnée, la nuit, que Vincent regarda le ciel nocturne et réalisa la beauté de notre galaxie. Il chercha alors à obtenir des informations sur les étoiles et planètes, afin de les retenir et de pouvoir se repérer lors de ses futures randonnées. Il lui faudrait donc une application portative (sur sa tablette) capable de lui permettre de mémoriser ces données. Il se tourna alors vers *Stellar*, qui ne nécessite aucune connexion internet, simple d'utilisation. Elle lui permettra d'apprendre des choses sur des étoiles et planètes et de dessiner les constellations qui vont lui servir à se repérer.



Sam Convient

Il faut préférer ce qui est impossible mais vraisemblable à ce qui est possible, mais incroyable.

Age : 24 ans

Profession : Développeur

Revenus : 31 000 brut/an

Situation : en couple, sans enfant.

Sam travaille dans la même entreprise depuis maintenant 3 ans. C'est un développeur de petits jeux, mais il lui arrive de faire du design. En dehors de son travail, il aime se former sur les nouvelles technologies et réalise de la veille.

Internet : utilisation quotidienne et intense

Sites préférés :

- docs.oracle.com
- stackoverflow.com

Applications préférées :

- Perfect Piano
- Duolingo

Passe-temps :

- Programmer et apprendre de nouveaux langages, apprendre des langues.
- Jouer du piano (depuis 6 mois), la musique plus généralement.

Profil technique :

- Maîtrise de nombreux langages de programmation orientés objets (Java, C++).
- Connaissance des langages du Web (HTML/CSS, PHP).
- Possède un téléphone Android Samsung Galaxy S10, une tablette graphique Microsoft et deux ordinateurs sous Linux.
- Passe 8h/j sur les écrans.
- Se forme en continu sur les nouveaux langages et nouvelles technologies.
- Améliore son anglais et apprend le Russe.

Un client demande à Sam de réaliser un petit jeu 2D en Java, de type RPG. Il devra incorporer un cycle jour/nuit pour ce projet, or Sam n'a aucune idée de l'image de fond à utiliser pour cela. Il va devoir créer une image de ciel nocturne de toutes pièces, et se tourne alors sur *Stellar*, qui va lui permettre de réaliser ce dont il a besoin, de manière assez simple :





Lucie Fer

Plus tard quand je serais grande, je serais maîtresse et j'enseignerais les maths !

Age : 8 ans

Profession : Sans emploi

Revenus : Aucun

Situation : Ecolière (CM1)

Lucie est encore écolière de primaire, mais cela ne l'empêche pas d'être très bonne élève. Elle a d'ailleurs sauté la classe de CE2. Toujours joviale, elle souhaiterait plus tard devenir maîtresse et enseigner les mathématiques. Elle aime aussi l'astronomie.

Internet : utilisation exceptionnelle sous autorisation des parents

Sites préférés :

- coloriage.info
- apprendremagie.com

Applications préférées :

- Google Chrome
- LibreOffice

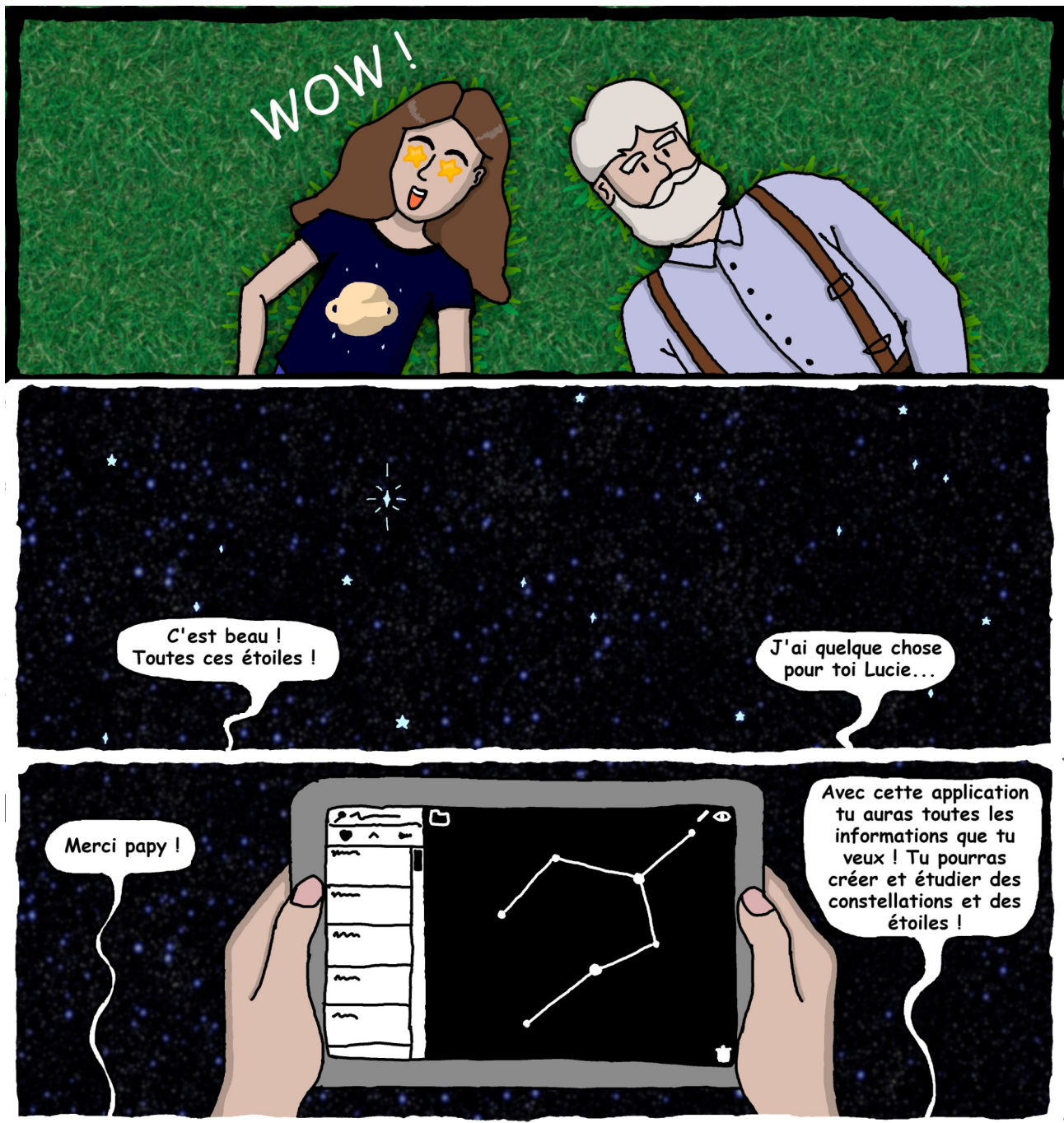
Passe-temps :

- Jouer avec ses ami.e.s (pendant les récréations).
- Apprendre à faire des tours de magie.
- Les coloriages et mots-croisés.
- Les sorties en famille.

Profil technique :

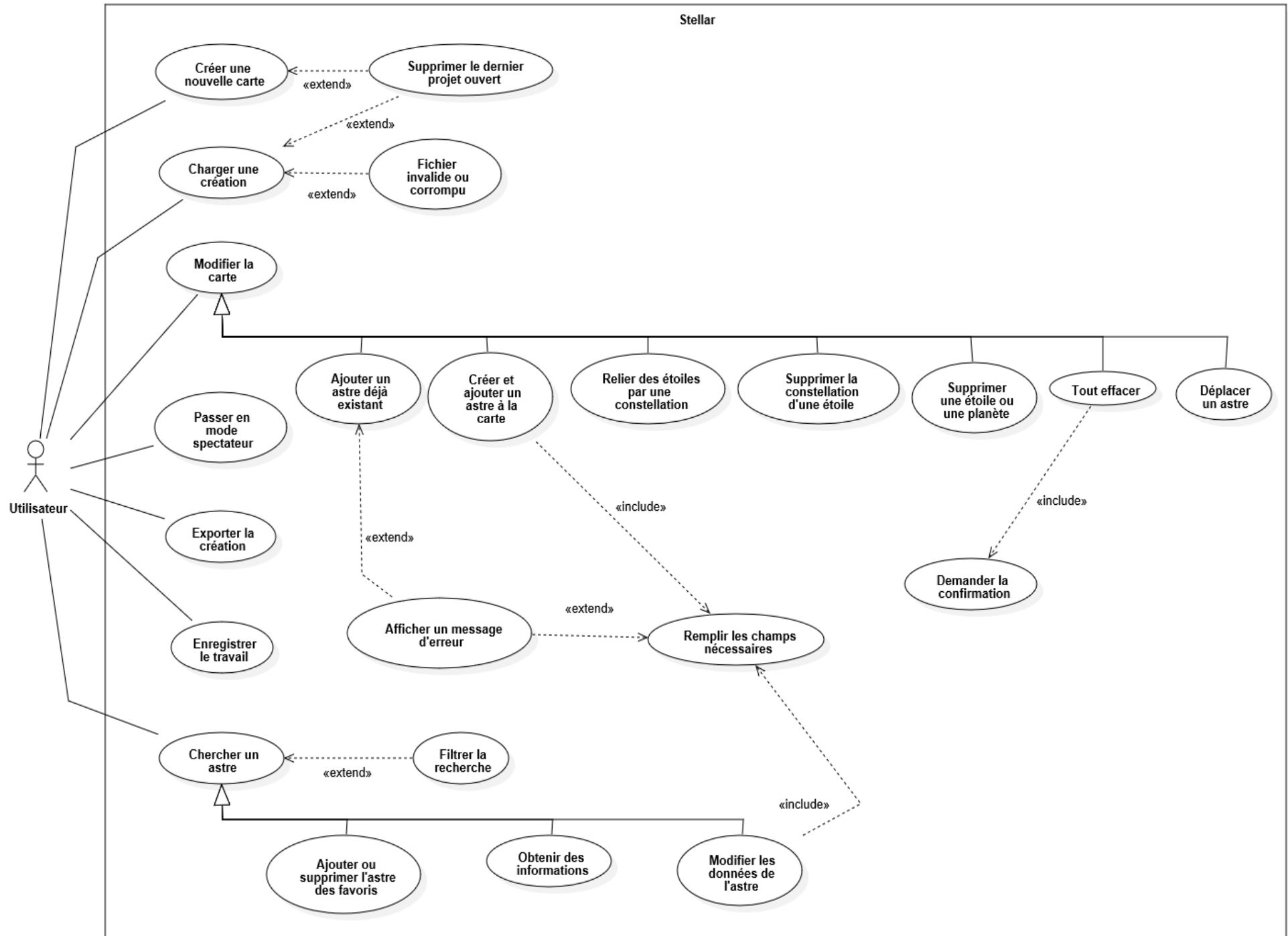
- Maîtrise la langue française et aussi l'Allemand (père d'origine Allemande).
- Connaît les rudiments des mathématiques (calculs, fractions, géométrie).
- A participé à plusieurs concours organisés par son école, dont le "Kangourou des mathématiques" qu'elle a remporté.
- Ne possède pas beaucoup de connaissances en informatique.
- S'intéresse à tout ce qui touche les sciences, comme le travail de sa mère (comptabilité).

Lors d'une soirée en extérieur avec son grand-père, Lucie regarde le ciel et se demande à quelle distance se trouve Sirius de chez nous... Alors son grand-père sort sa tablette avec l'application *Stellar*, de manière à ce que Lucie puisse obtenir toutes les informations qu'elle souhaite sur ces astres. Comme l'application ne nécessite aucune connexion internet, les parents n'ont pas besoin d'avoir recours à un contrôle parental !



Note : les visages de ces personnes ont été générés aléatoirement via le site <https://thispersondoesnotexist.com/>
Ils n'existent donc pas en vrai, il n'y a donc pas de droit d'image / de copyright.

1.3 Diagramme des Cas d'utilisation



1.4 Analyse du diagramme de cas d'utilisation

Nom	Charger une création
Objectifs	Charger un fichier de sauvegarde contenant les données de la création d'une carte (étoiles et planètes créées, informations sur celles-ci, constellations...).
Acteurs principaux	Utilisateur
Acteurs secondaires	Aucun
Conditions initiales	-L'utilisateur doit posséder un projet de l'application qu'il aura du enregistrer précédemment (qui contient des données de sauvegarde).
Scénario d'utilisation	<p>-L'utilisateur sélectionne « Ouvrir un fichier » depuis le menu dossier, ou réalise le raccourci clavier Ctrl + O.</p> <p>-Un message s'affiche comme quoi l'utilisateur peut ouvrir un projet, mais celui lui fera perdre les données du projet actuel, et lui conseille d'enregistrer le projet actuel avant d'en ouvrir un nouveau. L'utilisateur peut souhaiter de continuer et donc d'ouvrir le fichier, le cas « Supprimer le dernier projet ouvert » est alors réalisé. Si non, conditions de fin 2.</p> <p>-L'utilisateur est invité à parcourir les fichiers depuis son explorateur et sélectionne le fichier souhaité.</p> <p>-Si l'application détecte que le fichier ne contient pas les données d'un projet, ou si le fichier est corrompu, ou impossible à lire ou ouvrir, alors un message d'erreur s'affiche disant que le fichier n'a pu être traité (voir conditions de fin 3). Le cas « Fichier invalide ou corrompu » est réalisé.</p> <p>-S'il n'y a pas de problème, le projet est chargé en mémoire puis ouvert depuis l'application (conditions de fin 1).</p>
Conditions de fin	<p>1) Le projet a bien été chargé, l'utilisateur peut retrouver son travail (ses étoiles, planètes et sa carte), et travailler dessus.</p> <p>2) L'utilisateur a préféré annuler l'ouverture pour ne pas perdre son travail actuel. Il doit alors d'abord réaliser le cas « Enregistrer le travail » puis de nouveau réaliser le cas « Charger une création ».</p> <p>3) L'utilisateur a indiqué un mauvais fichier ou alors le fichier était corrompu ou déjà ouvert. Il doit alors veiller à fermer les fenêtres susceptibles d'utiliser ce document, et vérifier le fichier qu'il envoie au programme, puis de nouveau réaliser le cas « Charger une création ».</p>

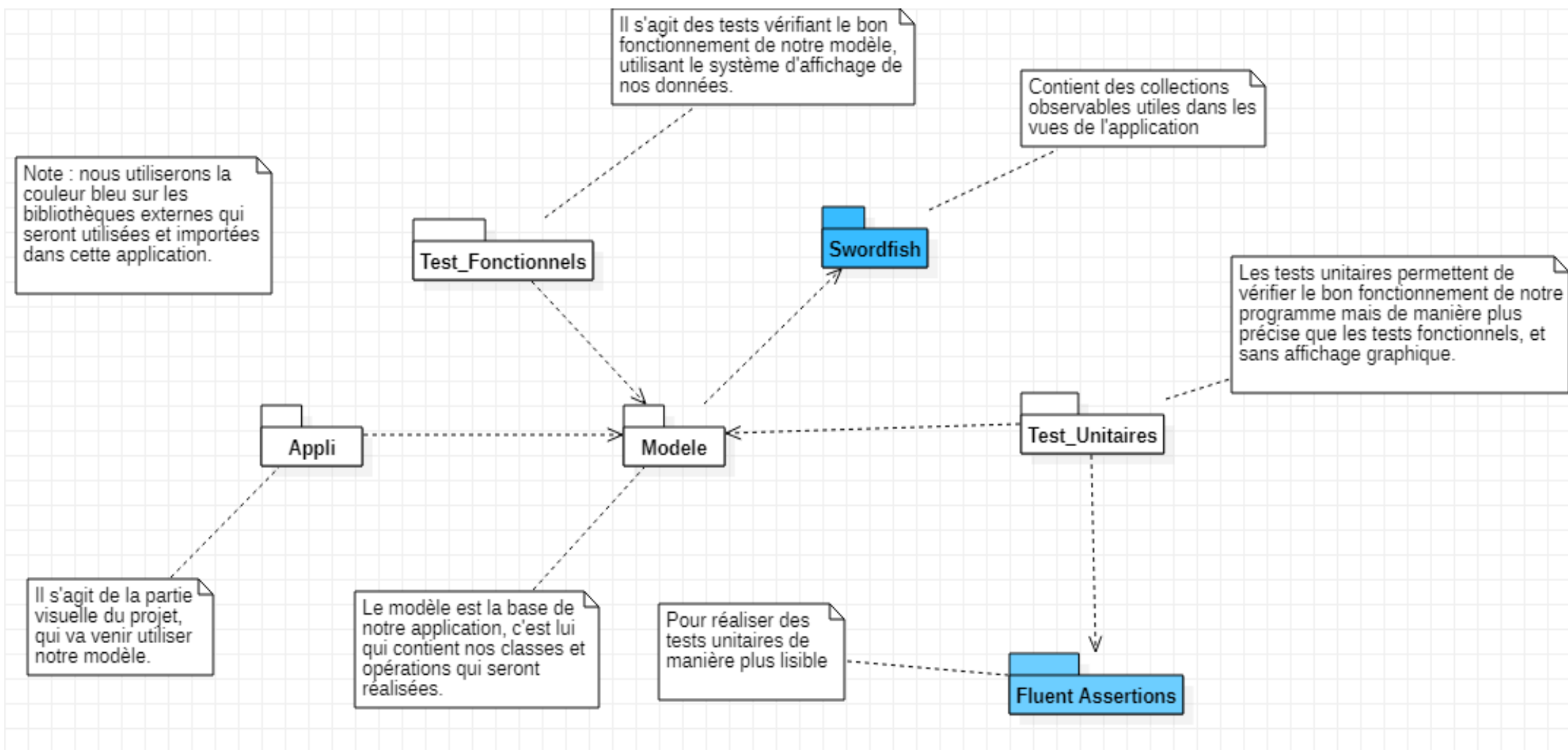
Nom	Supprimer une étoile ou une planète
Objectifs	Supprimer une étoile ou une planète de la carte, et les constellations (arêtes) qui lui sont reliées (si elle en a).
Acteurs principaux	L'utilisateur
Acteurs secondaires	Aucun
Conditions initiales	-L'utilisateur doit avoir au moins une étoile ou planète sur sa carte.
Scénario d'utilisation	<p>-L'utilisateur clique l'outil « suppression » (la gomme, en haut à droite).</p> <p>-Il peut annuler sa suppression, en cliquant de nouveau sur l'outil gomme (conditions de fin 2).</p> <p>-Il peut cliquer sur l'étoile ou la planète à supprimer. Si c'est une étoile et qu'elle possède des arêtes (= constellations) qui lui sont attachées, alors le premier clic sur l'étoile supprimera ces arêtes. Le second clic supprimera l'étoile. Si l'étoile n'a aucune arête, un clic suffira à supprimer l'étoile de</p>

	la carte. S'il s'agit d'une planète il suffira d'un seul clic également. L'utilisateur peut de nouveau cliquer sur l'outil gomme afin de quitter le mode suppression (conditions de fin 1).
Conditions de fin	1) L'étoile ou planète est alors effacée de la carte, mais aussi de la liste de gauche (elle disparaîtra) s'il s'agissait d'un astre personnalisé (= créé par l'utilisateur). Si c'est un astre intégré à l'application, il sera toujours visible dans la liste de gauche et pourra être de nouveau placé sur la carte. 2) L'utilisateur a annulé son action en désélectionnant l'outil gomme. S'il veut la supprimer, il a juste à réaliser de nouveau le cas « Supprimer une étoile ou une planète ».

Nom	Créer et ajouter un astre à la carte
Objectifs	Créer un astre personnalisé, avec le nom et informations que l'utilisateur souhaite, et le placer sur la carte.
Acteurs principaux	L'utilisateur.
Acteurs secondaires	Aucun.
Conditions initiales	Aucune.
Scénario d'utilisation	-L'utilisateur clique sur le bouton « Ajouter » (symbole étoile s'il veut ajouter une étoile, symbole rond s'il veut ajouter une planète) en haut à droite de la carte. -Il clique ensuite sur une zone de la carte pour pouvoir ajouter son étoile ou sa planète qui sera placée à cette position. Un pop-up s'ouvre et le cas « Remplir les champs nécessaires » doit être réalisé. Pour cela, l'utilisateur doit remplir les champs obligatoires notés d'un astérisque. S'il clique sur la croix ou sur « Annuler », alors l'astre sera supprimé de la carte et l'ajout ne sera pas réalisé (conditions de fin 2). -Sinon, il peut cliquer une « Valider ». S'il a choisi un nom d'astre déjà existant dans le logiciel, alors le cas « Afficher un message d'erreur » est réalisé. Dans ce cas, un pop-up s'affiche et lui demande de changer de nom. S'il clique sur la croix ou sur « Annuler », alors l'astre sera supprimé de la carte et l'ajout ne sera pas réalisé (conditions de fin 3). -Sinon, l'ajout se fait correctement (conditions de fin 1).
Conditions de fin	1) L'ajout a été réalisé sur la carte. L'astre a été créé et est maintenant visible depuis la liste de gauche. Il peut modifier les champs de celui-ci s'il le souhaite. 2) L'utilisateur a annulé son ajout. L'astre n'a pas été ajouté. S'il veut ajouter un astre, il doit alors réaliser de nouveau le cas « Créer et ajouter un astre à la carte ». 3) L'utilisateur a annulé son ajout suite à la réalisation du cas « Afficher un message d'erreur ». Il peut alors vérifier depuis la liste (en utilisant la barre de recherche), si le nom de l'astre déjà existant est un astre personnalisé. Si non, il lui faut trouver un autre nom pour son étoile qui ne peut pas avoir le même nom qu'une étoile déjà présente dans le système, et réaliser de nouveau le cas « Créer et ajouter un astre à la carte ». Si oui, alors il peut réaliser le cas « Supprimer une étoile ou une planète », qui supprimera son astre personnalisé précédent, puis le cas « Créer et ajouter un astre à la carte », qui créera un astre avec le nom qu'il souhaitait lui donner.

II – Conception Orientée Objet

2.1 Diagramme de paquetage



2.2 Analyse du diagramme de paquetage

Nous utiliserons dans ce modèle une bibliothèque de classes, nommée *Modele*, qui contiendra les classes, méthodes, interfaces, énumérations et autres opérations qui seront utilisées par notre *appli*.

L'*appli* justement sera en réalité la vue, sur laquelle il sera possible de naviguer, et de réaliser des actions qui auront été justement définies dans le *Modele* comme expliqué précédemment. Il s'agit de l'aspect visuel de notre projet.

Les tests fonctionnels se déroulent sur application console, et permettent de tester les différentes méthodes, collections, et opérations définies dans le *Modele*. Ils utilisent le système d'affichage, ce qui permet de vérifier le bon fonctionnement de notre *Modele*. Ils se décomposent en plusieurs classes qui contiennent elles-mêmes plusieurs méthodes, permettant de vérifier le fonctionnement d'éléments spécifiques.

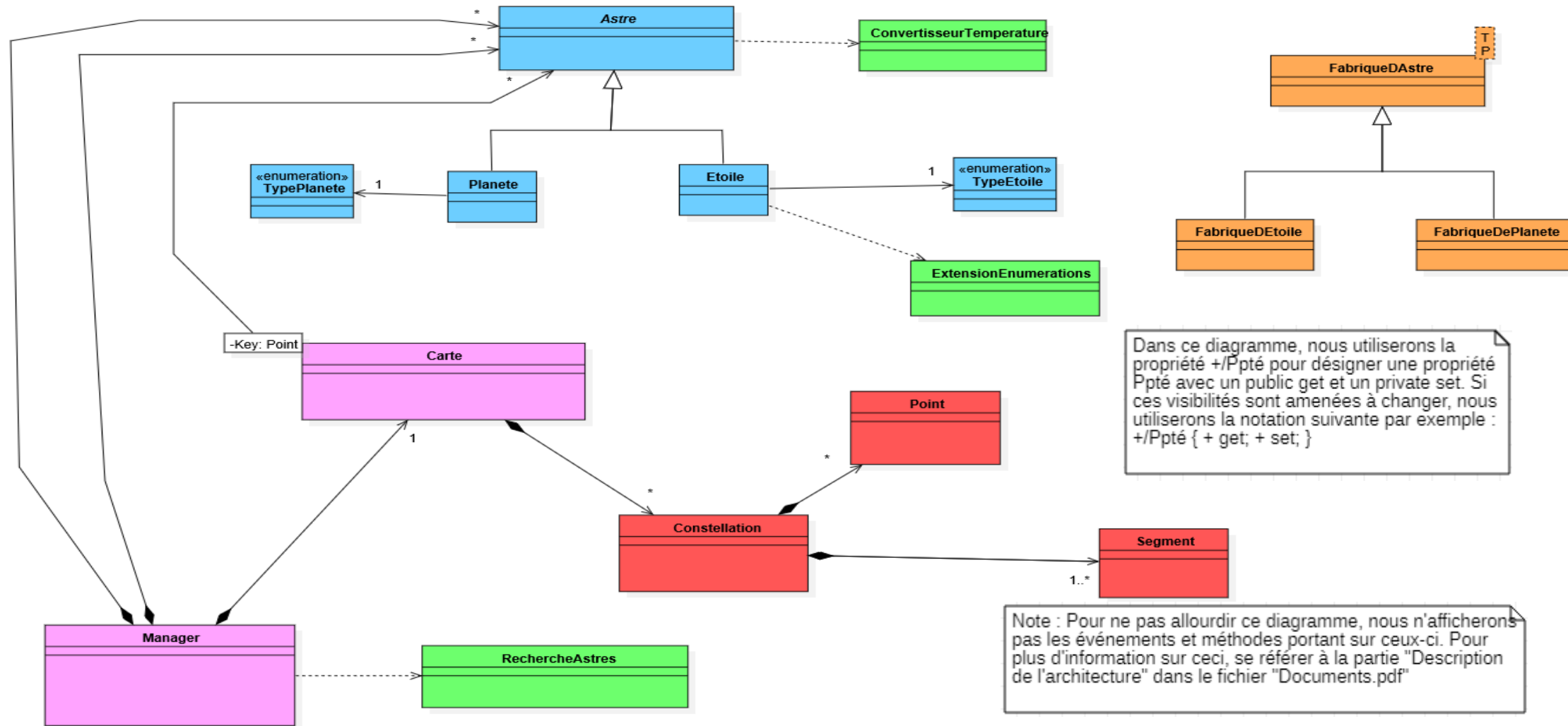
Les tests unitaires sont similaires aux tests fonctionnels, mais n'utilisent pas d'interface console. Ils vérifient simplement la cohérence des données, contenues dans des instances, classes, après des instanciations ou opérations, telle que la suppression, l'ajout, la modification, etc. de notre *Modele*. Ces tests se décomposent également en plusieurs classes, permettant chacune de tester diverses instanciations ou opérations.

Les tests unitaires utiliseront *FluentAssertion*, qui est un *Nuget* (une bibliothèque externe), qui permet de réaliser des tests de manière plus lisse, plus facile à lire, à créer, et à comprendre.

Enfin notre *Modele* utilisera le *Nuget Swordfish* qui contient des collections observables qui auront une utilité dans la vue, pour l'affichage de nos données.

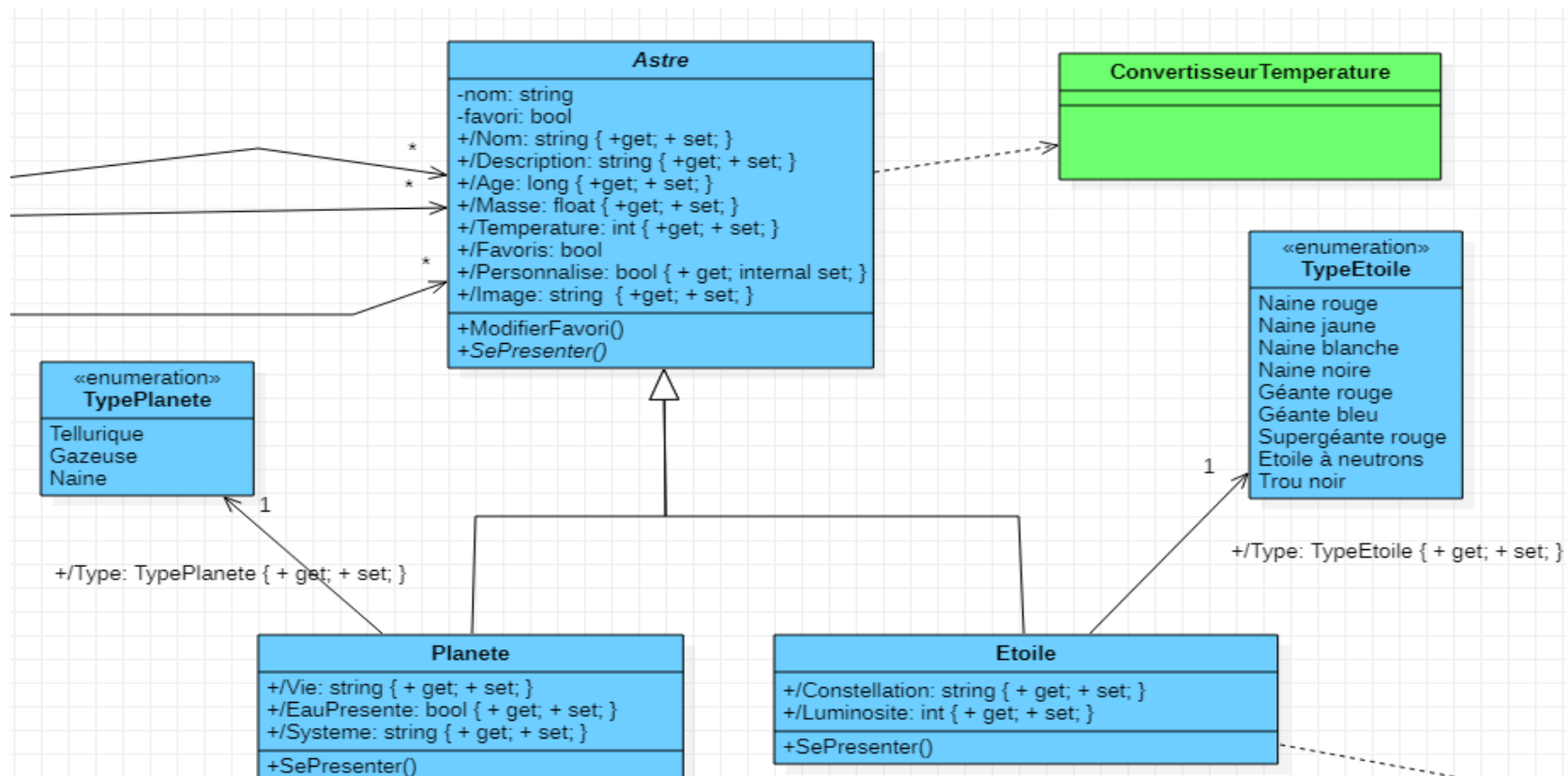
2.3 Le diagramme de classes

Il est possible de voir ici notre diagramme de classe de manière simpliste (les méthodes, attributs, propriétés ont volontairement été enlevés). Les différentes parties colorées vont être détaillées sur la suite de ce document.



2.4 Analyse du diagramme de classes

2.4.1 Le cœur de l'application : les Astres



Nous allons ici nous concentrer sur la partie bleu. Il s'agit de la source de notre application. Tout d'abord, nous allons nous concentrer sur la classe abstraite *Astre* et ses descendantes.

Un astre est un objet quelconque de l'Univers, il est caractérisé par un nom, un âge (qui peut être assez grand), une masse, une température, et éventuellement une description. Dans le cadre de cette application nous aurons deux types d'astres : ceux déjà présents dans le logiciel, et ceux créés par l'utilisateur. Il y aura donc un booléen qui indiquera si un astre est personnalisé ou non.

Un astre pourra être noté comme étant dans les favoris de l'utilisateur, d'où le booléen permettant d'indiquer cela, il y aura donc une méthode permettant de changer l'état du favori de l'astre. Il possédera également une image.

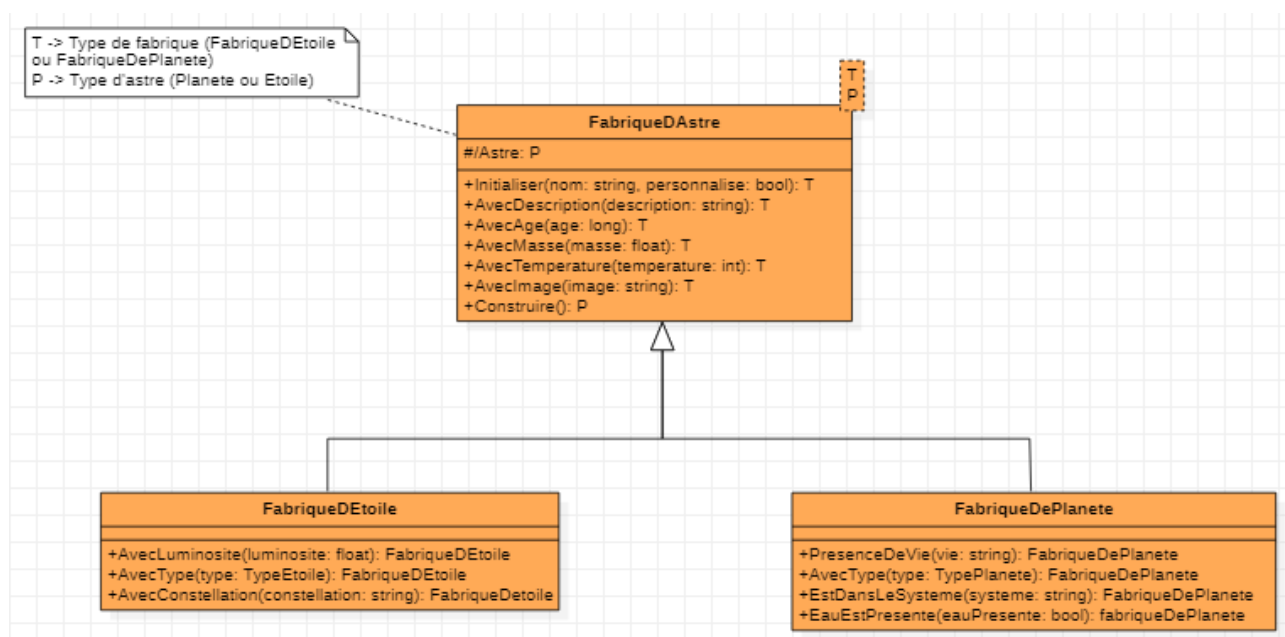
Or un *Astre* est un objet assez abstrait, d'où l'intérêt ici de ne pas pouvoir en instancier un. En revanche, il y a deux liens d'héritage qui partent d'*Astre*, et permettent de spécifier qu'il peut s'agit d'une étoile ou d'une planète. Chacune possède ses propres caractéristiques, notamment une luminosité et une constellation pour l'étoile, ainsi qu'une indication sur la présence d'eau ou de vie, et le système stellaire dans le cas d'une planète.

Chaque spécification de l'*Astre* possède un *Type*. Une étoile peut être une naine blanche, une naine jaune, un trou noir..., il y a donc une énumération qui permet d'énumérer ces différents types. Dans le cas de la planète, c'est similaire. Elle peut être tellurique, gazeuse ou naine.

Chaque astre pourra se présenter, et donner son type.

Avec cette configuration il est facile de maintenir le code et de rajouter de nouvelles spécifications d'astres par la suite.

2.4.2 Les fabriques d'Astres



Nos constructeurs d'astres étaient assez gros suite à l'héritage, et nous voulions pouvoir créer des astres de manière un peu plus *fluent*, un peu plus lisible et propre.

Nos classes *Astre*, *Etoile* et *Planete* possédant beaucoup de champs, il arrivait qu'on ne s'y retrouve plus lors de l'instanciation d'objets via ces constructeurs. Nous avons donc utilisé ce patron de conception afin de réaliser des instanciations plus lisibles, plus *fluent*.

Nous avons également une relation d'héritage entre les fabriques, un astre restant un objet abstrait, il nous faut spécifier si nous allons créer une étoile ou une planète. Par conséquent, la *FabriqueDAstre* sera utilisée par la *FabriqueDEtoile*, et par la *FabriqueDePlanete*.

La fabrique d'astre utilise donc la généricité. Elle prend deux types T et P lors de l'instanciation. T correspond au type de fabrique (donc dans notre cas, on veut soit avoir une *FabriqueDEtoile*, soit une *FabriqueDePlanete*), et P au type d'astre que l'on souhaite créer (donc dans la même logique, soit une *Etoile*, soit une *Planete*).

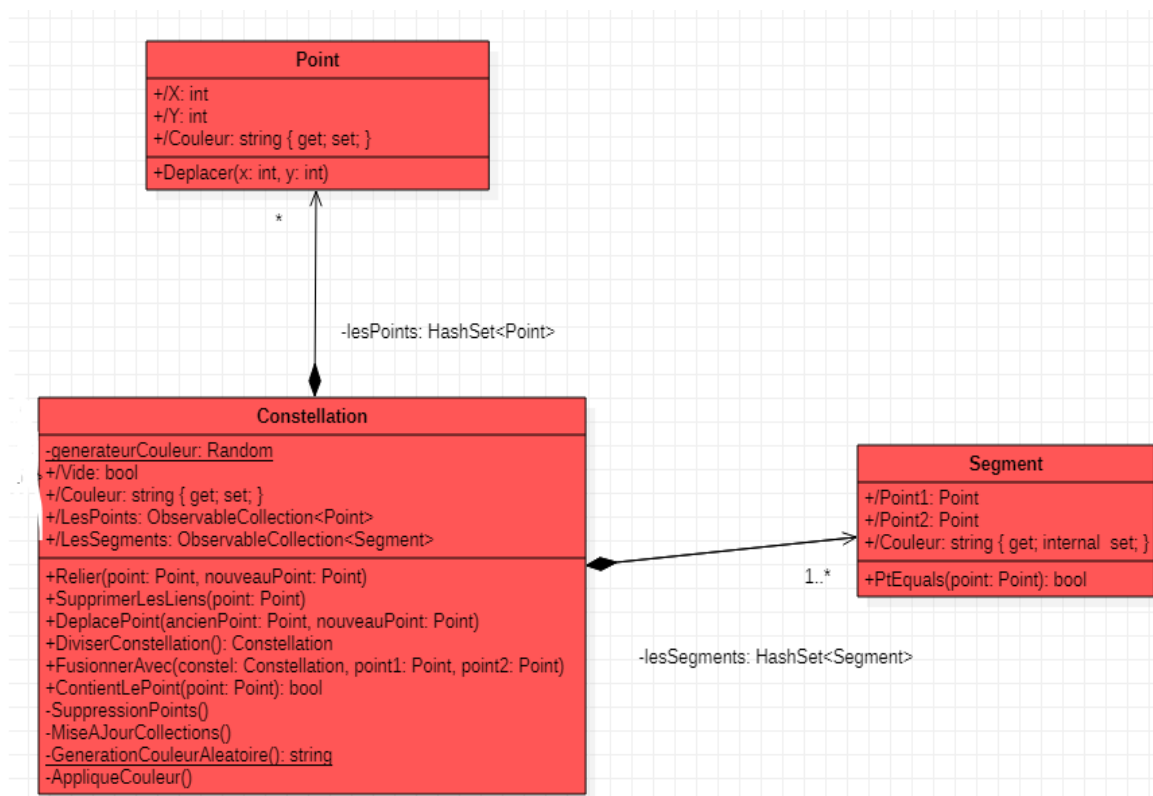
L'astre contenue dans la fabrique d'astre est protégée afin que ses classes filles y ait accès. Il est toujours de type P (*Etoile* ou *Planete*).

Par la suite, on retrouve une méthode d'initialisation qui permet d'instancier un astre en fonction de son type générique P . Concrètement on vient créer soit une étoile soit une planète. Il existe diverses méthodes permettant d'ajouter des paramètres à notre astre P , que l'on retourne sous forme de fabrique T . Cela permet de chaîner les méthodes entre elles, et faire des appels successifs d'ajouts de paramètres, de manière plus épurée qu'avec un gros constructeur.

Les classes *FabriqueDEtoile* et *FabriqueDePlanete* permettent d'ajouter les méthodes propres aux attributs de l'astre en question. Il ne sera possible de renseigner la constellation que pour une étoile par exemple.

Enfin la méthode de construction dans la fabrique d'astres permet de retourner cette fois-ci notre astre créé. Ces fabriques sont décrites de manière plus précise dans la partie *Description de l'architecture*.

2.4.3 Les constellations



Il doit être possible dans notre application de créer des constellations. Ces constellations peuvent donc s'apparenter à un graphe : c'est un ensemble de points reliés entre eux par des segments.

Nous avons donc créé une classe *Point*. Un point est caractérisé par ses coordonnées en

abscisses et en ordonnées, et il est possible de le déplacer. Il possède également une couleur (jaune par défaut).

Un *Segment* est caractérisé par deux points : le point de départ et le point d'arrivée. Tout comme le point, il possède une couleur. On considère que deux segments sont égaux s'ils possèdent les deux mêmes points, mais pas forcément dans le même ordre (le segment n'est pas orienté). Il possède également une méthode permettant de savoir si un point est contenu dans le segment.

Enfin une *Constellation* est composée d'une collection de points (sous forme de *HashSet* pour éviter les doublons), et de segments (de même que pour les points). Une constellation possède une couleur qui sera appliquée à l'ensemble de ses segments afin de faire ressortir sur la carte.

Elle possède également deux collections de points et de segments, qui sont observables. Ce sera utile pour la mise à jour de l'affichage dans la vue.

Une constellation est composée au minimum de deux points et d'un segment.

L'utilisateur doit pouvoir relier des étoiles entre elles, et même déplacer les points. Cette classe possède donc diverses méthodes, notamment une lui permettant de relier deux points, de déplacer un point, de savoir si un point est contenu dans la constellation.

Lorsque que deux points sont reliés, il peut arriver que deux constellations soient amenées à être fusionnées ensemble, d'où la méthode présente ici.

Enfin, il doit être possible de supprimer les segments d'un point en particulier, et donc effacer ce point de la collection (la méthode *SupprimerLesliens*). Il y a une méthode permettant d'effacer les points qui ne sont reliés à rien.

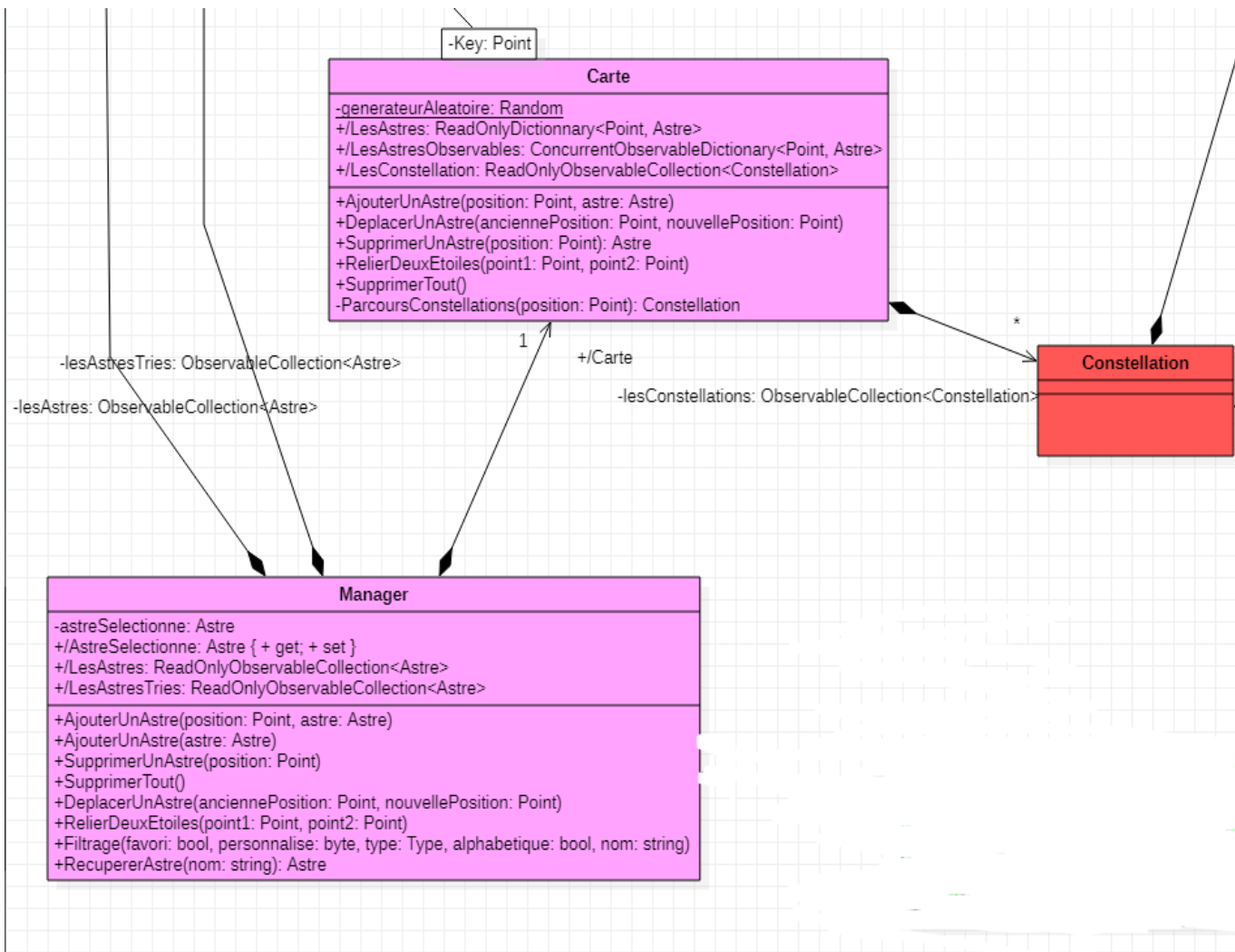
Pour finir, la méthode *DiviserConstellation* permet de réaliser un parcours en largeur sur la constellation afin de ne garder qu'une composante connexe, et retourne une nouvelle constellation contenant l'(les) autre(s) composante(s) connexe(s).

La méthode de mise à jour permet de s'assurer qu'aucun doublon ne se trouve dans les collections observables.

Une constellation possède une couleur qui sera appliquée à l'ensemble de ses segments, et cette couleur sera initialisée via un générateur aléatoire et la méthode statique lors de l'instanciation.

Enfin, une méthode permettra d'appliquer cette couleur sur l'ensemble des segments.

2.4.4 Le Manager et la Carte



La classe *Carte* est la classe qui correspond à la partie éditeur de notre application (elle contient les données de cette partie de l'application). C'est elle qui est chargée de modifier les données concernant les constellations, points, qui sont ajoutés / supprimés / déplacés sur la carte.

Elle contient une liste de constellations, conformément à ce qui a été dit précédemment. Elle possède ainsi une méthode pour relier des étoiles. Attention, il est possible de ne relier que des étoiles, pas des planètes.

Elle est également caractérisée par un dictionnaire clé/valeur avec pour clé un *Point*, et pour valeur un *Astre*, ce qui permet de rapidement retrouver un astre à partir de sa position sur la carte.

Elle possède une version observable des deux collections citées précédemment.

Il est donc possible d'ajouter un astre sur la carte, avec un nouveau point de position. Il est également possible de déplacer un astre à une nouvelle position. Un astre peut aussi être supprimé, ce qui peut entraîner des modifications des constellations. Enfin, il est possible de tout effacer.

La classe *Manager* est la classe centrale de notre *Modele*. C'est elle qui gère la navigation

dans notre fenêtre en appelant les méthodes nécessaires. Elle contient également les données de l'application car elle possède une liste d'astres, ce sont les astres contenus dans l'application mais aussi ceux qui ont été créés par l'utilisateur. Elle possède également une Carte sur laquelle l'utilisateur peut ajouter et modifier des données.

Il est donc naturellement possible de modifier cette liste d'astres en ajoutant des données, en en supprimant, en en modifiant certaines, d'où les méthodes d'ajouts, de suppression, etc.

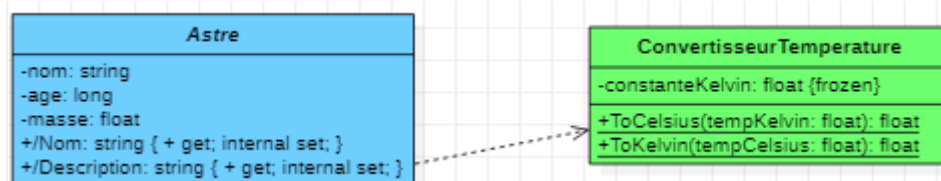
Il contient une version observable de la collection précédente, mais également une version observable d'une collection d'astres triés. Elle correspond à la collection qui sera affichée dans l'application lors des recherches.

Les méthodes de déplacement et de reliement sont déléguées à la carte.

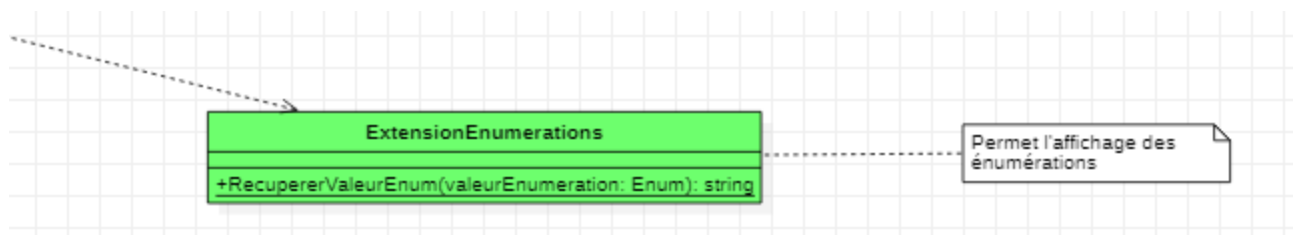
Le Manager contient également une méthode de Filtrage, qui permet d'appeler la classe utilitaire de tri en conséquence afin de mettre à jour la liste d'astres triés.

Il possède un getter spécial qui permet de retourner un booléen indiquant si un astre fournit en paramètre se trouve bien dans la collection d'astres.

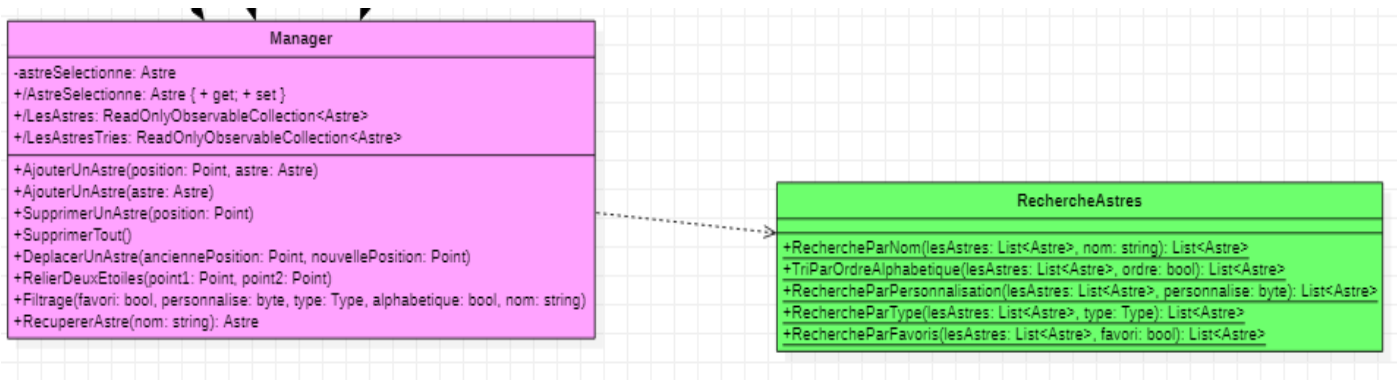
2.4.5 Les classes utilitaires



La classe *ConvertisseurTemperature* permet de convertir la température en différentes unités, et permet d'obtenir un rendu et un affichage plus cohérent. Il est utilisé au sein de la classe *Astre*.



La classe *ExtensionEnumerations* est aussi une classe utilitaire permettant de récupérer une énumération mais avec des espaces, et donc de pouvoir l'afficher de manière à avoir un meilleur rendu textuel, d'avoir des caractères spéciaux et espaces.



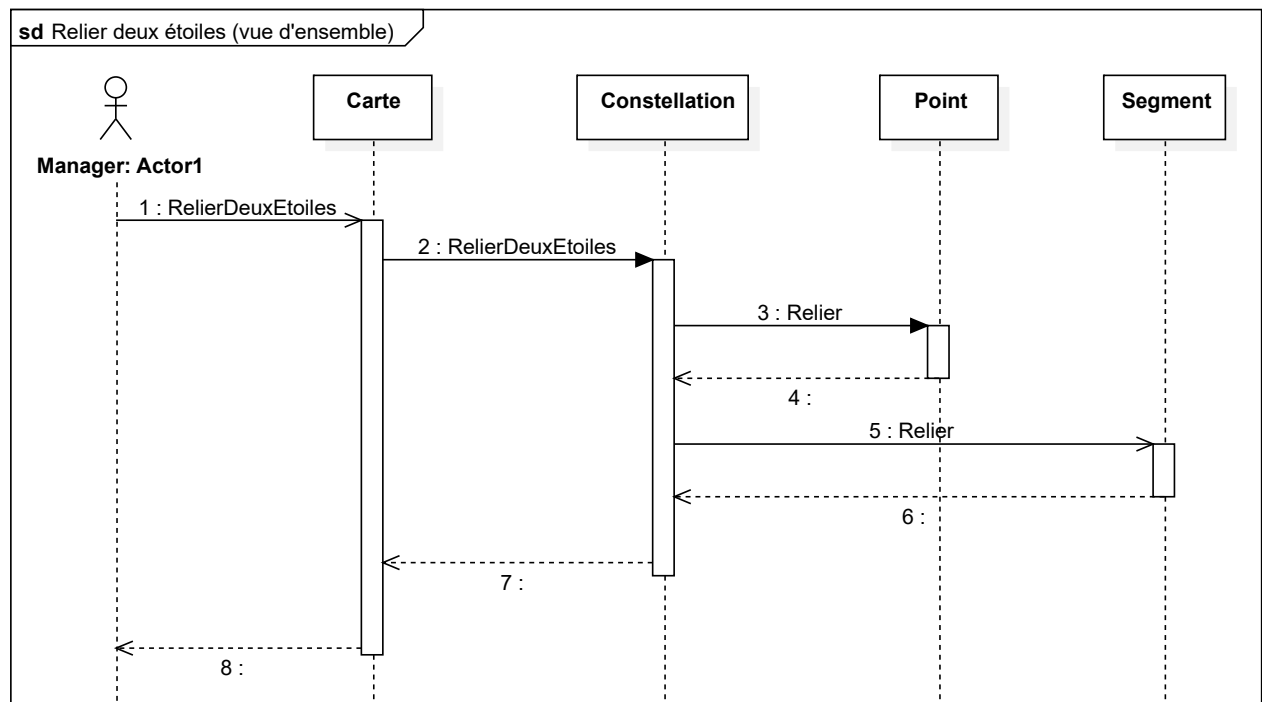
Enfin la classe *RechercheAstres* permet d'effectuer divers tris et filtrages. Elle est utilisée par la manager dans la méthode de *Filtrage*, pour pouvoir trier les différents astres en fonction des envies de l'utilisateur.

Note : Pour voir l'ensemble du diagramme, on peut le visualiser à cette adresse (fichier StarUML) : <https://forge.clermont-universite.fr/svn/stellar/trunk/documentation/Diagramme%20de%20classes.mdj>

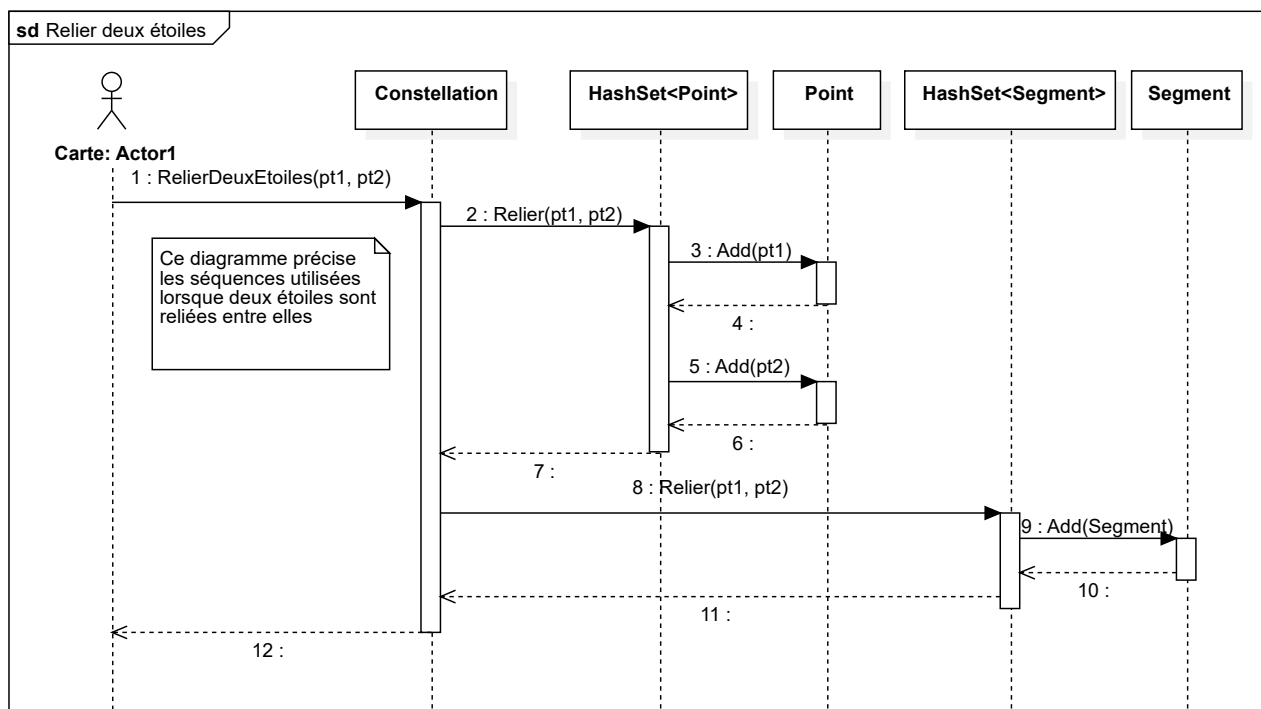
(attention, un espace entre le « de » et le « % » a tendance à être ajouté, vérifier le lien).

2.5 Diagrammes de séquences

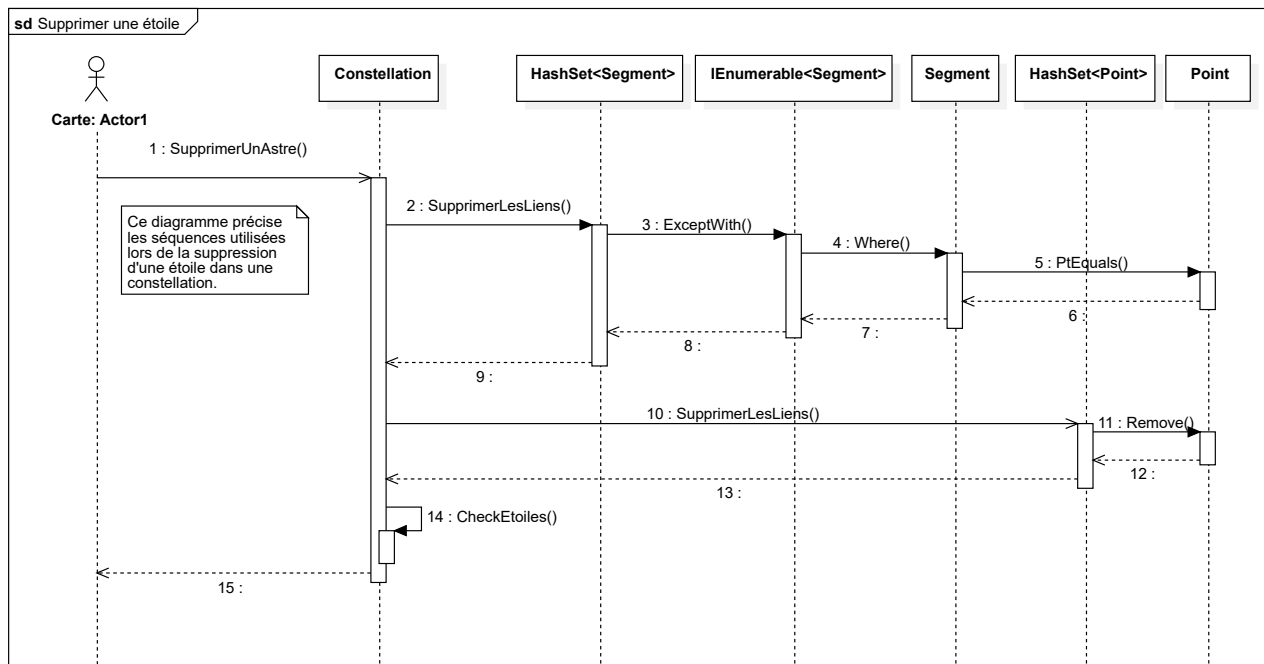
2.5.1 Relier deux étoiles (vue d'ensemble)



2.5.2 Relier deux étoiles



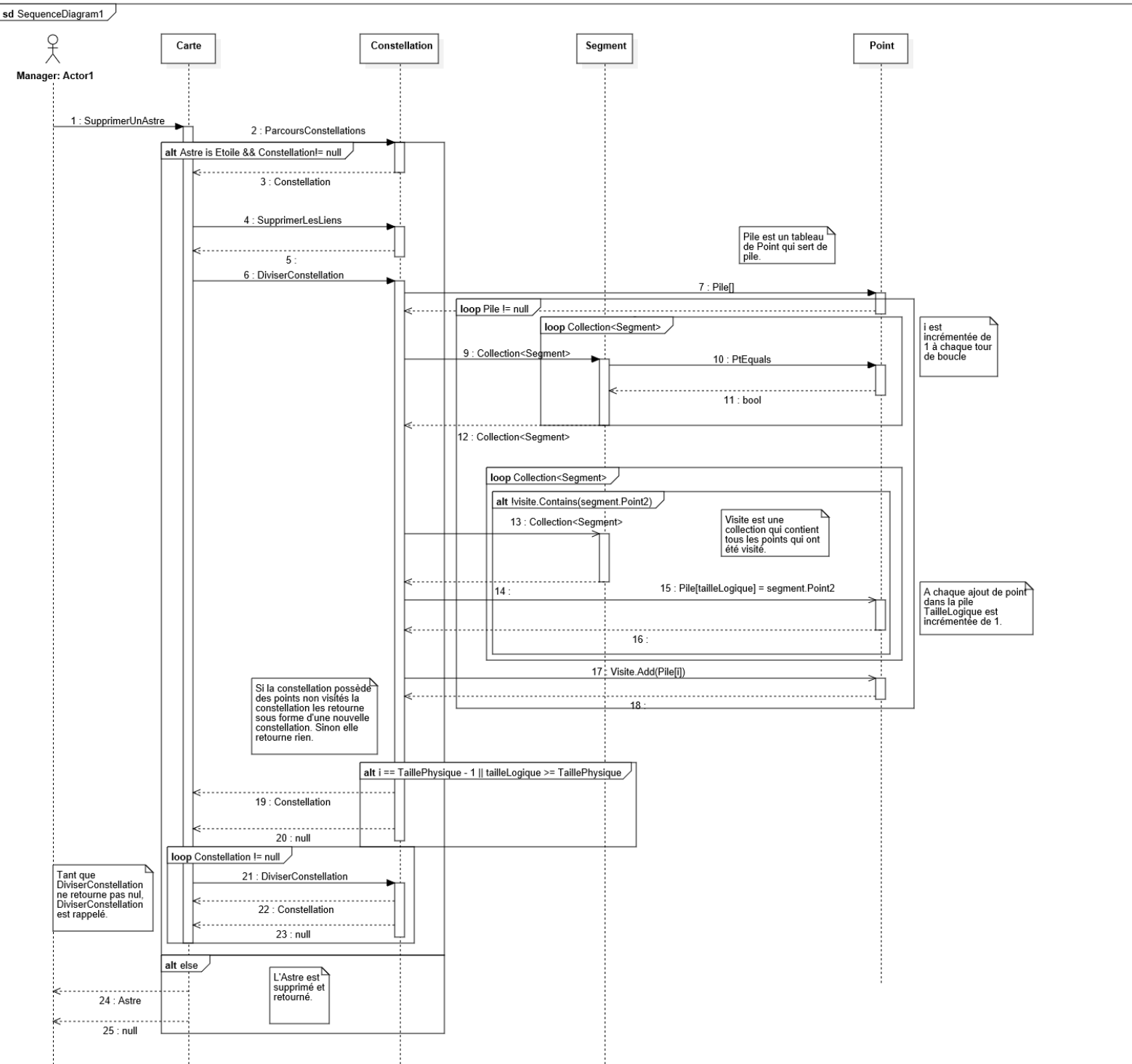
2.5.3 Supprimer une étoile dans une constellation



2.5.4 Supprimer un Astre

Note : nous avons essayé de raccourcir ce diagramme, mais l’affichage n’est pas optimal, il est assez gros et de taille plutôt carré. S’il est trop peu lisible, il est possible de le retrouver sous forme de fichier SVG, à l’adresse suivante :

<https://forge.clermont-universite.fr/svn/stellar/trunk/documentation/Diagramme3.svg>



2.6 Analyse des diagrammes de séquences

2.6.1 Relier deux étoiles (vue d'ensemble)

Lorsque qu'un utilisateur veut relier deux étoiles entre elles. Il interagit avec la Carte constitué de plusieurs Constellations. Il clique sur deux étoiles, on distingue alors plusieurs cas. Les étoiles peuvent relier deux Constellations différentes (les constellations seront donc fusionnées), une étoile appartenant à une constellation peut relier une autre étoile n'appartenant à aucune constellation (l'étoile orpheline est englobée dans la constellation l'autre étoile), ou bien les deux étoiles ne se trouvent dans aucune constellation (une nouvelle constellation est créée). La Constellation ou les Constellations contenant les étoiles sont sélectionnées puis les étoiles sont reliés entre elles.

2.6.2 Relier deux étoiles

Lorsque la méthode *RelierDeuxEtoiles* de Carte est appelée, deux points sont passés en paramètres : pt1 et pt2. Carte appelle la méthode *Relier* de la constellation sélectionnée ou créée une Constellation si nécessaire. Ces deux points sont ajoutés dans un *hashset* de points. Ils sont en suite utilisés pour créer un nouveau segment qui sert à relier les étoiles entre elles. Le segment est en suite ajouté dans un *hashset* de segments. Les *HashSet* permettent de constituer une Constellation d'étoiles.

2.6.3 Supprimer une étoile dans une constellation

Lorsque la méthode *SupprimerUnAstre* de carte est appelée. Le point à supprimer est passé en paramètre. Si le point est une étoile et qu'elle est dans une constellation, ses liens avec les autres étoiles sont supprimés. Les liens (ou segments) à supprimer sont retournés par une commande *LINQ*, puis ils sont supprimés du *HashSet* de segments de la constellation. Le point à supprimer est supprimé du *HashSet* de points de la constellation. Enfin la méthode *CheckEtoiles* de la constellation est appelée. Elle permet de supprimer tout les points du *HashSet* de points de la constellation qui n'ont plus de liens avec les autres points (qui sont donc isolés).

2.6.4 Supprimer un Astre

Fonctionnement général :

Si l'astre sélectionné est une planète ou une étoile ne faisant pas partie d'une constellation alors l'astre est tout simplement supprimé et retourné. En revanche si l'astre est une l'étoile qui est dans une constellation (voir le diagramme *Supprimer une étoile dans une constellation*) alors ses liens sont supprimés et la méthode *DiviserConstellation* est appelée. Puis à la fin de l'exécution de la méthode *SupprimerUnAstre*, *null* est retourné (on ne veut pas supprimer l'étoile, on a effacé les liens de la constellation).

Fonctionnement de *DiviserConstellation* :

Lors de la conception de notre projet nous nous sommes heurtés à un problème de taille. Lorsqu'une étoile dans une constellation est supprimée, elle peut dans certains cas créer plusieurs constellations. Les constellations sont visuellement séparées mais pour le programme ces constellations n'en forment qu'une. Cela crée donc une incohérence dans notre *Modele*.

Nous avons considéré les constellations comme des graphes non-orientés et effectué un algorithme de parcours en largeur afin de déterminer si une constellation est complète (ou connexe).

Cet algorithme permet de s'assurer que l'on a bien une seule et unique composante connexe dans chaque constellation.

Lorsque *SupprimerUnAstre* de carte est appelée, la méthode appelle *DiviserConstellation* dans la Constellation sélectionnée. Cette méthode effectue un parcours en largeur: l'algorithme du parcours en largeur utilise un tableau de Points servant de pile. La pile indique les Points à « visiter ». Quand la pile indique un point à « visiter » les autres points reliés par des segments sont ajoutés à la pile seulement s'il n'ont pas été visités et qu'il ne sont pas déjà dans la pile. Pour « avancer » de point en point l'algorithme utilise les segments les reliant, ils permettent de former un lien entre tous ces points, car ils contiennent bien un point de départ et d'arrivée.

Lorsque tout les points ont été ajoutés à la pile, l'algorithme passe au point suivant et ajoute les nouveaux points accessibles et ainsi de suite. Les points qui ont été visités sont ajoutés dans une collection de points servant à constituer la constellation visitée. L'algorithme s'arrête lorsqu'il ne peut plus visiter de point ou que la pile a atteint sa taille maximal (égale au nombre total de points de la constellation). Si l'algorithme est terminé et qu'il reste des points non visités dans la constellation alors le reste est retourné sous forme d'une nouvelle constellation.

Cette nouvelle constellation retournée peut ne pas être connexe, elle peut elle-même contenir plusieurs composantes connexes. C'est pour cela que dans Carte, il y a une boucle chargée d'appeler la méthode *DiviserConstellation* tant que la constellation retournée n'est pas nulle.

2.7 Description écrite de l'architecture

2.7.1 Architecture globale du programme

Notre projet est décomposé en 4 sous-projets (voir plus loin pour le détail). Concernant l'architecture générale, nous avons une bibliothèque de classes (dans notre cas, c'est *Modele*), une application fenêtrée (=visuelle), ici *Appli*. Nous avons enfin deux projets de tests (unitaires et fonctionnels) qui servent à tester le bon fonctionnement de notre *Modele*. *Modele* contiendra les classes utiles à notre application, sur laquelle *Appli* pourra se reposer pour les actions qu'il sera possible de réaliser.

Notre *Modele* s'articule autour de notre classe Manager qui vient déléguer certaines actions à la classe Carte (les détails sont fournis plus loin). Les autres classes permettent de hiérarchiser les données de l'application, et stipulent comment elles seront traitées.

2.7.2 Dépendances

Comme précisé précédemment, ce projet se décompose en 4 autres parties bien distinctes. Les voici :

- **La Bibliothèque de classes *Modele***

C'est elle qui contient toutes les classes et énumérations utilisées par notre programme, telles que présentées avant la partie sur le diagramme de classes. Elle se décompose elle-même en plusieurs *espaces de noms* différents qui précisent le rôle de chacune de ces classes :

L'espace de nom *Geometrie*

Notre projet est constitué d'éléments géométriques tels que des points et segments, qui sont utilisés dans l'application pour apporter une cohérence visuelle, et plus généralement, pour afficher les données de notre projet.

Il nous fallait donc réaliser ces classes, car c'est sur celles-ci que va s'appuyer les classes plus importantes comme Constellation ou encore Carte décrites ci-après.

Le nom de cet espace suggère qu'il reste possible à n'importe quel instant de rajouter des figures géométriques plus évoluées telles que des rectangles, cercles ou bien plus encore.

L'espace de nom *Espace*

Ce dernier constitue la base qui sera utilisée par l'espace de nom *Modele* décrit plus loin. Il s'agit des données mêmes qui seront gérées dans notre application, la source même de l'affichage de notre application.

Comme son nom l'indique, il contient les données relatives aux astres qui sont des objets abstraits et quelconques de l'Univers. Ainsi, nous avons réalisé une spécification de ces astres en des objets concrets, tel que des *Etoiles* ou encore des *Planetes*, mais il est possible de rajouter bien d'autres éléments, tel que des nébuleuses, des amas d'étoiles, ou encore des satellites.

Des énumérations permettent de spécifier les types de ces astres de manière plus concrète.

Enfin, la classe Constellation repose entièrement sur l'espace de nom géométrie décrit précédemment. En effet, une constellation est une composante connexe composée de segments et de points.

L'espace de nom *Utilitaire*

Comme son nom le laisse entendre, cet espace de nom contient les classes utilitaires de notre programme, qui seront généralement appelées par les classes racines de notre *Modele* (voir plus bas, le *Manager*).

Il est par exemple utilisé pour des conversions de températures, d'affichage des énumérations ou même pour réaliser divers tris et filtrages.

L'espace de nom *Modele*

Du même nom que le projet, il laisse sous-entendre qu'il s'agit de l'élément fondateur du projet. C'est lui qui est chargé de gérer les données, d'effectuer des tris, de modifier / ajouter / supprimer des éléments de collections, et bien d'autres encore.

Il se compose de fabriques utilisées pour instancier des manière simpliste, les données de notre application.

Les classes *Carte* et *Manager* sont les piliers de l'application, chargées de gérer les différentes actions que l'utilisateur pourra réaliser sur notre application. Elles contiennent également les collections relatives aux données de l'espace de nom *Espace*.

- **L'application visuelle *Appli***

Ce sous-projet est en réalité la façade de notre application. Il s'agit de la partie *front*, celle qui est visible et utilisée par un utilisateur lambda. Il s'agit du produit fini recherché.

Cependant elle ne peut fonctionner de manière autonome. Elle utilise le *Modele* comme définit précédemment et s'appuie sur les données contenues dans le *Manager* et la *Carte* comme décrit ci-dessus. On a donc une dépendance de l'*Appli* vers le *Manager*, car elle ne peut fonctionner sans.

- **Le projet de Tests Unitaires**

Ce sous projet est en réalité un projet de tests comme son nom l'indique. Il contient des tests unitaires, tests précis et non visuels qui s'appuient sur les classes créées dans le *Manager*. Ils visent à vérifier le bon fonctionnement des collections, constructeurs, méthodes...

Ils ont besoin d'une dépendance vers le *Modele* pour fonctionner.

- **Le projet de Tests Fonctionnels**

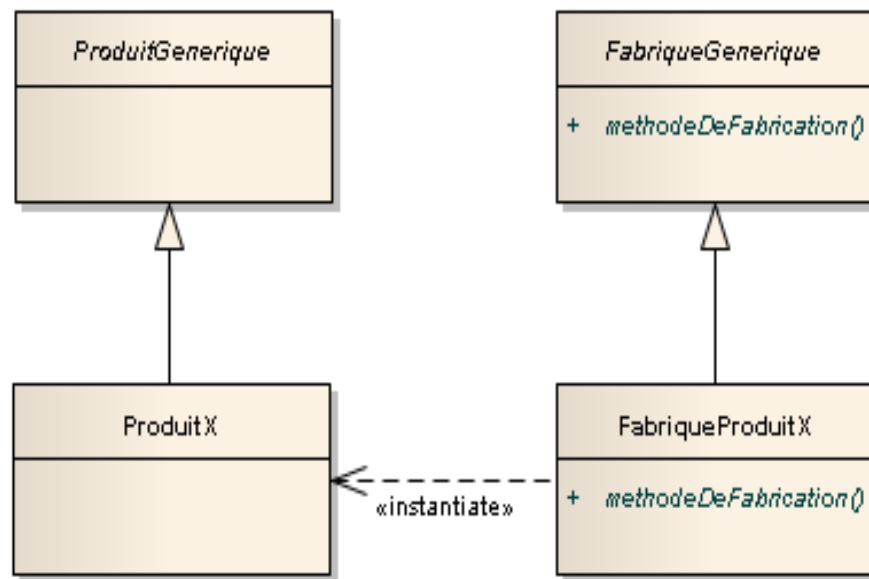
Ce dernier projet est similaire au précédent. Il s'agit également d'un projet de tests, mais qui sont cette fois-ci visuels, via un affichage console.

Tout comme pour les tests unitaire, ce projet de tests a besoin d'une référence sur *Modele* pour tester le bon fonctionnement de ses diverses classes.

2.7.3 Patrons de conception

Notre projet utilise divers patrons de conception afin de standardiser notre programme, et que celui-ci soit plus facilement maintenable, compréhensible et réutilisable.

Le premier a déjà été nommé, il s'agit de **la fabrique**.



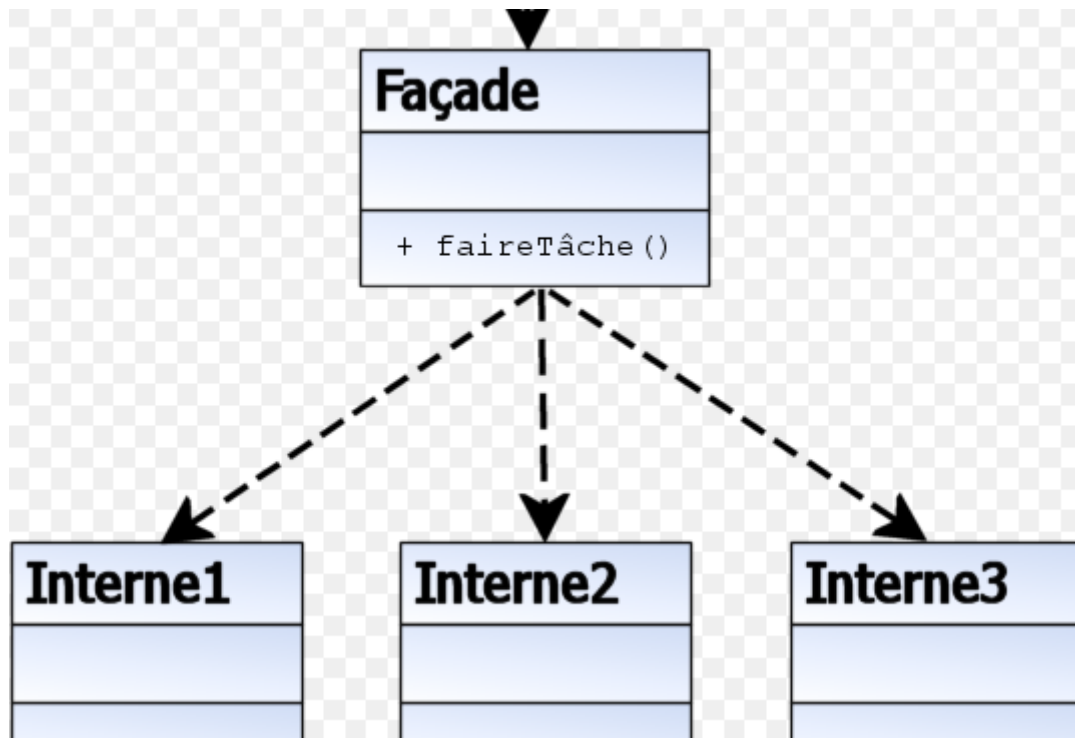
La fabrique permet d'instancier nos astres de manière plus *fluent*, comme elle a été présentée dans la partie portant sur la description du diagramme de classes.

Elle permet de créer une interface pour la création d'un objet, et facilite grandement la lisibilité, et évite les longs constructeurs. Dans notre cas nous avons plusieurs fabriques car il est impossible d'instancier directement des astres, il faut passer par des fabriques de planètes ou par des fabriques d'étoiles.

La classe créée n'est jamais notifiée qu'elle a été créée par une fabrique, elle ne connaît pas même son existence.

C'est un patron de conception très utile, il permet véritablement *d'interfacier* la création de nos astres, de la rendre plus simple.

Le second patron se nomme **la façade**.

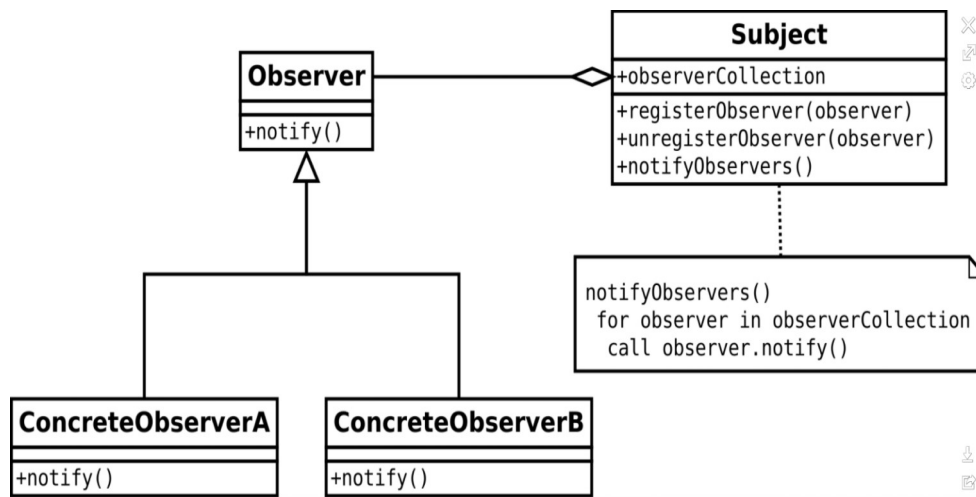


Comme expliqué dans le diagramme de classes, le Manager sera la pièce maîtresse de notre *Model*. C'est lui qui vient gérer les plupart des actions, cependant, il est aussi là pour déléguer. La plupart des actions effectuées lors de l'ajout d'un astre ou de sa suppression, le fait de relier des étoiles, etc. auront un impact sur la carte qui sera modifiée à son tour. Il ne fait qu'appeler la méthode correspondante depuis la Carte.

On observe donc ici un patron de conception bien précis : la façade.

La façade ici, la Manager, gère les données principales et délègue ses actions à la carte. Il s'agit de la pièce maîtresse de l'application. A aucun moment les sous-classes appelées par le Manager ne font référence à la façade, elles ne le connaissent même pas. Elle permet de simplifier un système de gestion complexe en une seule classe qui va venir gérer le code de manière autonome.

Enfin le dernier patron utilisé est l'Observateur.



Il est utilisé pour notifier les les objets qui dépendent d'un autre objet, lorsque ce dernier change d'état. Concrètement, il permet de gérer des évènements, et de gérer le couplage de manière à la limiter aux phénomènes à observer (d'où son nom).

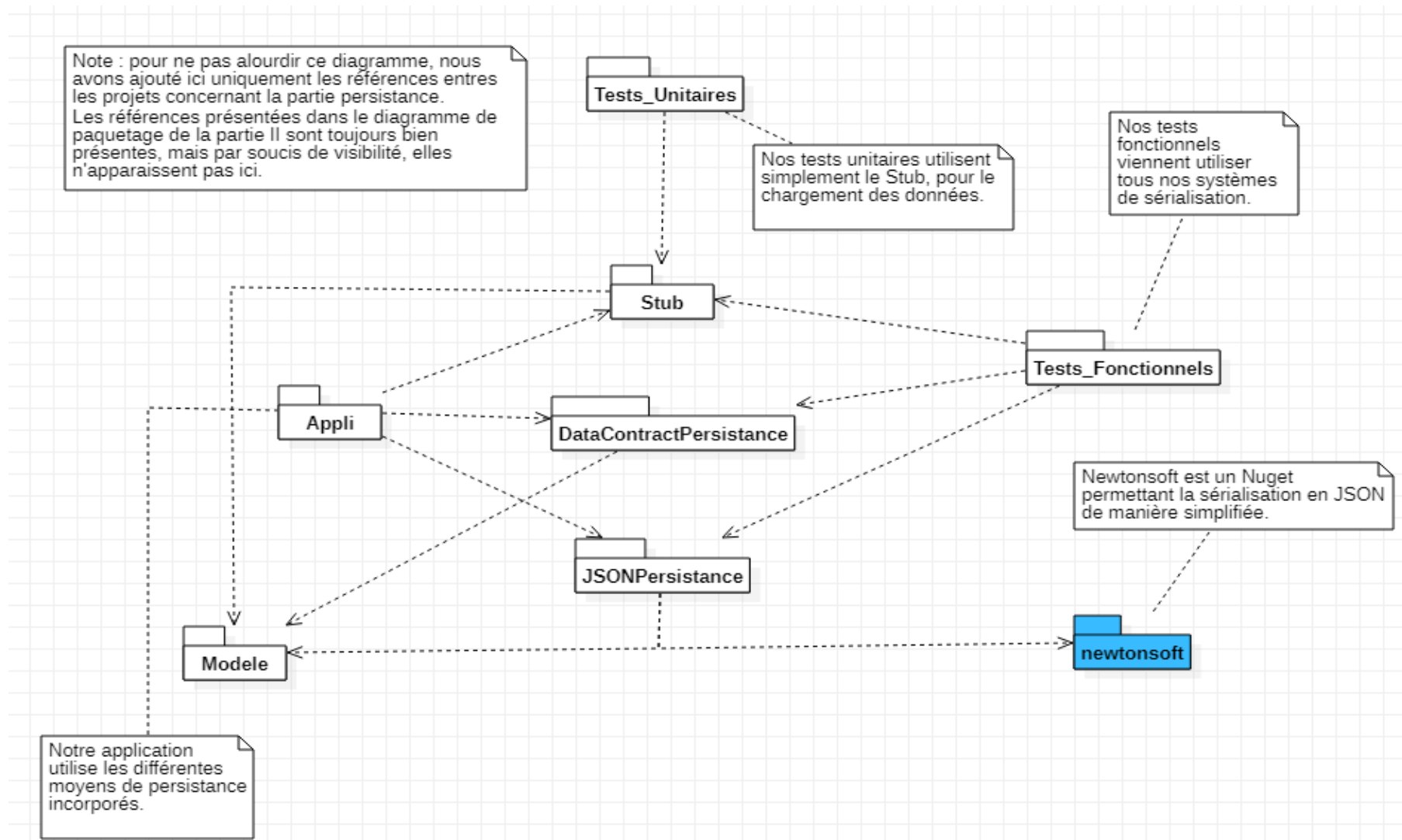
Dans notre cas il est utilisé pour notifier notre vue (*l'Appli*), qu'un élément change d'état, et cela permet donc à notre application de mettre à jour le visuel de la fenêtre. Nous l'avons implémenté à deux endroits : dans la classe abstraite *Astre* et dans le *Manager*.

Ce patron utilise les observateurs concrets qui viennent faire les mises à jour pour que le visuel de l'application soit modifié. En revanche l'observé (dans notre cas, *Astre* et *Manager*), n'est pas responsable de ces changements.

Dans notre projet, il permet de notifier par exemple que le champ favori d'un astre a été modifié, afin que la vue puisse l'afficher de manière différente. Il est aussi utilisé pour mettre à jour la collection d'astres triés du *Manager* lorsque celle-ci varie (quand l'utilisateur modifie les filtres), afin que la nouvelle collection puisse être affichée. Pour l'utiliser, on implémente *INotifyPropertyChanged* et on appelle la méthode *OnPropertyChanged* quand des données sont modifiées.

III – Projet Tuteuré

3.1 Diagramme de paquetage sur la persistance



3.2 Analyse du diagramme de paquetage

Comme indiqué sur les notes du diagramme juste au-dessus, les références dans le diagramme de paquetage de la partie II ont été conservées. Elles n'apparaissent pas ici.

Sur ce diagramme de paquetage, c'est la partie persistance qui est ici mise en avant. Nous avons donc 3 modes de persistance différents. Le *Stub* tout d'abord, qui est seulement un moyen pour charger nos instances (il ne permet pas la sauvegarde).

Le second projet correspond au mode de persistance du *DataContractPersistance*. Il permet de sérialiser les données en XML.

Enfin le *JSONPersistance* est un projet permettant la sérialisation des données en JSON. Ce mode de persistance est un peu spécial, car il utilise un *Nuget* nommé *Newtonsoft*, qui permet la sérialisation en JSON de manière simplifiée. Il possède donc une dépendance sur ce dernier.

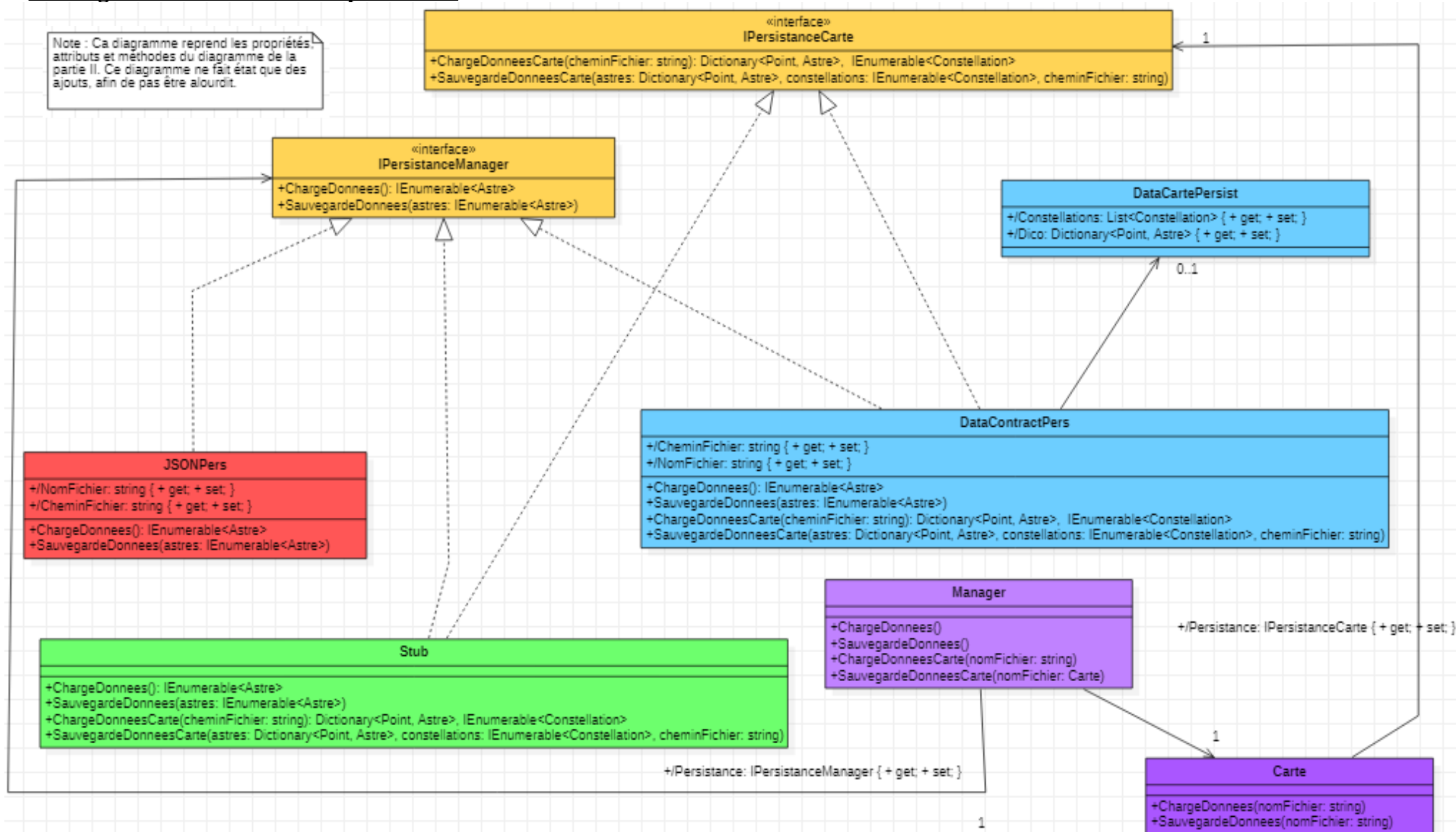
Les trois modes de persistance qui viennent d'être énoncés possèdent une référence sur le *Modele*, car ils doivent connaître les objets et les collections qui vont sérialiser, qui se trouvent directement dans le *Modele*.

Nos *Tests_Fonctionnels* possèdent maintenant des références sur les trois de ces modes de persistance, car nous nous en servons afin de pouvoir vérifier le bon fonctionnement de ces modes.

Nos *Tests_Unitaires* possèdent uniquement une référence vers le *Stub*, car cela leur permet directement de charger les données de manière simplifiée. Nous ne testons pas le fonctionnement des autres modes de persistance avec les tests unitaires.

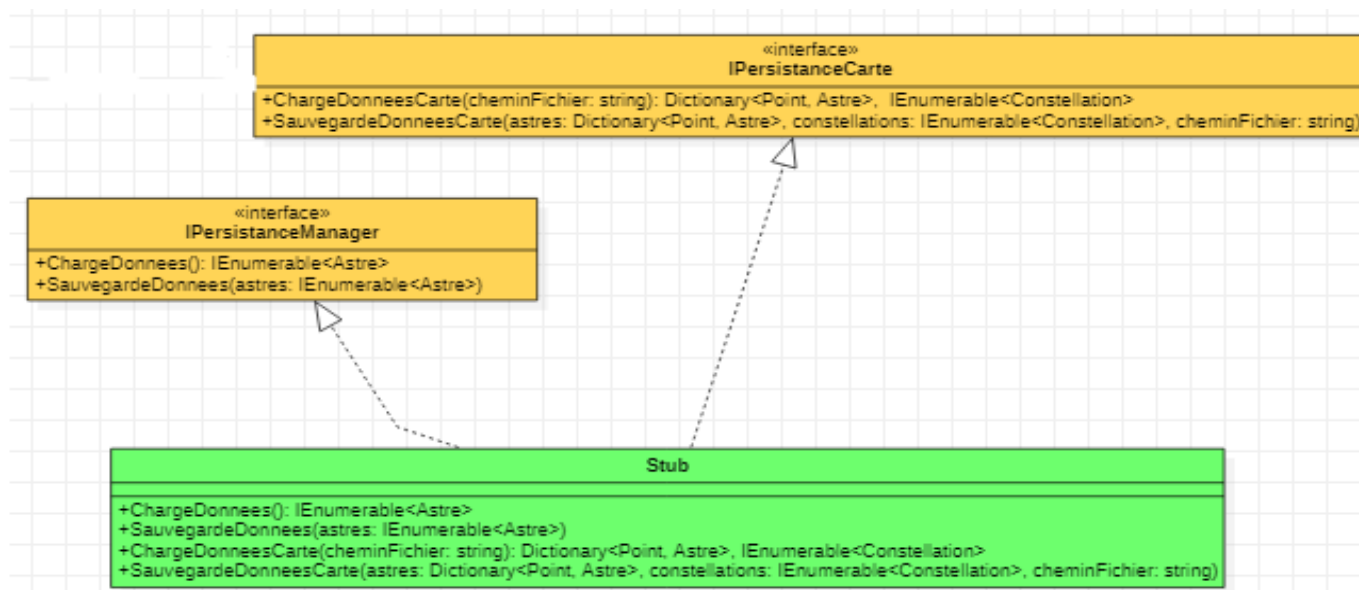
Enfin, notre Appli utilise les trois modes de persistance, car c'est pour elle qu'ils ont été conçus, pour la sauvegarde et le chargement des données de notre application visuelle. Elle doit donc les connaître pour pouvoir les utiliser.

3.3 Diagramme de classes sur la persistance



3.4 Analyse du diagramme de classes

3.4.1 Le Stub



Nous avons tout d'abord nous intéresser à la partie jaune. Il s'agit de nos interfaces *IPersistenceCarte* et *IPersistenceManager*.

Comme toutes les interfaces, elles définissent un patron, un contrat que devra respecter la classe qui l'implémente.

Comme décrit dans le précédent diagramme de classes plus-haut, nous avons une classe *Manager* qui contient une liste d'astres, qui correspondent aux données de notre application. Nous avons également une classe *Carte* qui correspond à la partie « éditeur » de notre application. Elle contient un dictionnaire de points et d'astres (afin de retrouver plus facilement un astre à partir de son point), ainsi qu'une liste de constellations.

Il faut donc pouvoir charger et sauvegarder les données de notre *Manager* (autrement dit, notre liste d'astres), et de même avec la *Carte* (notre dictionnaire et notre liste).

Nous aurons donc deux systèmes de persistance différents, fonctionnant de manière séparée. Nous avons donc conçu deux interfaces. Chacune de ces interfaces possède une méthode de chargement et une méthode de sauvegarde, en fonction de la classe concernée (comme dit précédemment, depuis la carte, on doit pouvoir enregistrer et charger notre dictionnaire et notre liste, et depuis la *Manager*, uniquement la liste d'astres).

Les interfaces ne précisent pas réellement comment vont se réaliser ces actions. Elles doivent être implémentées par une autre classe, et c'est le cas ici avec le *Stub* (en vert).

Le *Stub* vient simuler un système de persistance, même si ce n'en est pas réellement un. Comme il implémente les deux interfaces, ils contient les 4 méthodes que contenaient ces interfaces.

La méthode de chargement de la liste du *Manager* consiste simplement à placer toutes nos

données en « dur », dans une liste, puis à retourner cette liste. La méthode de sauvegarde ne fait rien (il est impossible de sauvegarder des données dans du code).

La méthode de chargement de la Carte rend simplement des collections vides (nous aurions voulu créer une carte avec quelques éléments dessus, placés aléatoirement, mais nous n'avons pas eu le temps). La méthode de sauvegarde ne fait rien non plus.

Le *Stub* vient donc uniquement charger les données du Manager (la liste d'astres), et c'est tout. Les modes de persistance vont être décrits par la suite.

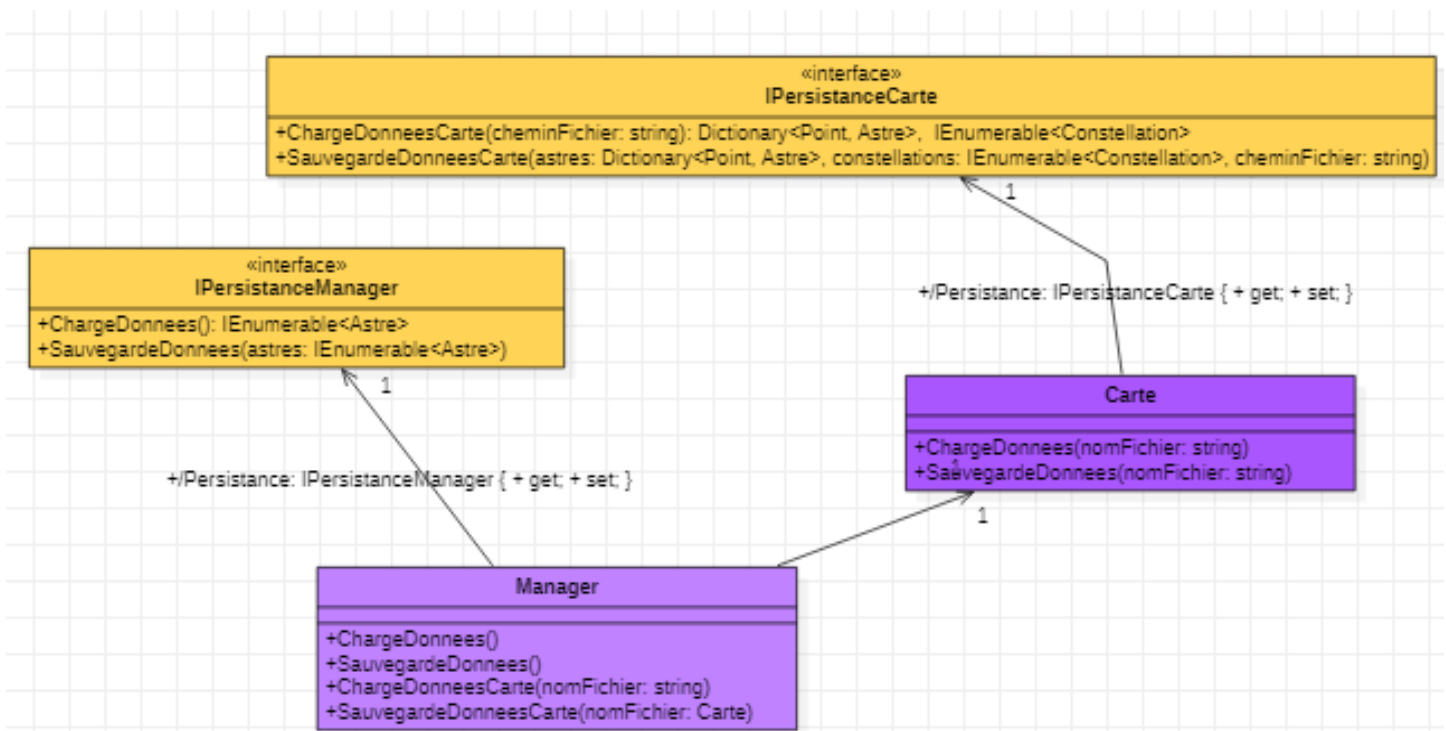
3.4.2 Aparté sur le patron de conception

Nous avons réalisé plusieurs modes de persistance comme décrit dans le diagramme de classes page 33. Or, pour réaliser une telle chose, il va nous falloir préciser dans les classes dont on souhaite persister des données (ici le Manager et la Carte), par quel mode de persistance nous allons le faire.

Dans ce cas de figure, nous cherchons à mettre en place un système de persistance générique dans nos classes, afin de pouvoir le changer facilement par la suite, sans casser tout ce qui a été fait dans les différents modes de persistance ou dans les classes qui l'utiliseront.

Pour cela, un patron de conception bien précis existe et nous l'avons utilisé, il s'agit de la Stratégie.

Ce patron de conception permet de pouvoir changer des algorithmes rapidement, sans casser ou modifier le code des classes appelantes et appelées.



Dans notre cas, nous utilisons le patron stratégie dans le Manager. Le Manager va donc posséder un mode de persistance, il possédera donc un *IPersistenceManager*.

Il s'agit d'un type générique, et donc un *IPersistenceManager* peut représenter n'importe quel mode de persistance du moment que celui-ci implémente *IPersistenceManager*. Cela va donc permettre de mettre en place ce patron de conception.

Il possède un setter public, comme cela il sera simple de changer le mode de persistance. On pourra passer d'une persistance JSON, en XML, juste en une ligne de code, sans modifier notre implémentation, notre Manager, et nos méthodes de persistance.

Le Manager possède donc tout naturellement une méthode de chargement et de sauvegarde qui vont venir seulement appeler les méthodes de chargement et de sauvegarde de notre *IPersistenceManager*. Ainsi, quel que soit le mode de persistance, nos méthodes de chargement et de sauvegarde seront toujours accessibles et fonctionnelles.

Pour la Carte nous utilisons le même schéma : notre Carte possède un *IPersistenceCarte*. Comme pour le Manager, celui-ci est renseigné dans le constructeur ou peut-être modifié via le mutateur.

La Carte possède deux méthodes : une de sauvegarde et une de chargement, qui appellent simplement les méthodes de sauvegarde et de chargement de notre *IPersistenceCarte*. Les données retournées par la méthode de chargement permettent la mise à jour des données de la carte, et pour la sauvegarde on fait l'inverse : on envoie les données de la carte en vue de la sérialisation.

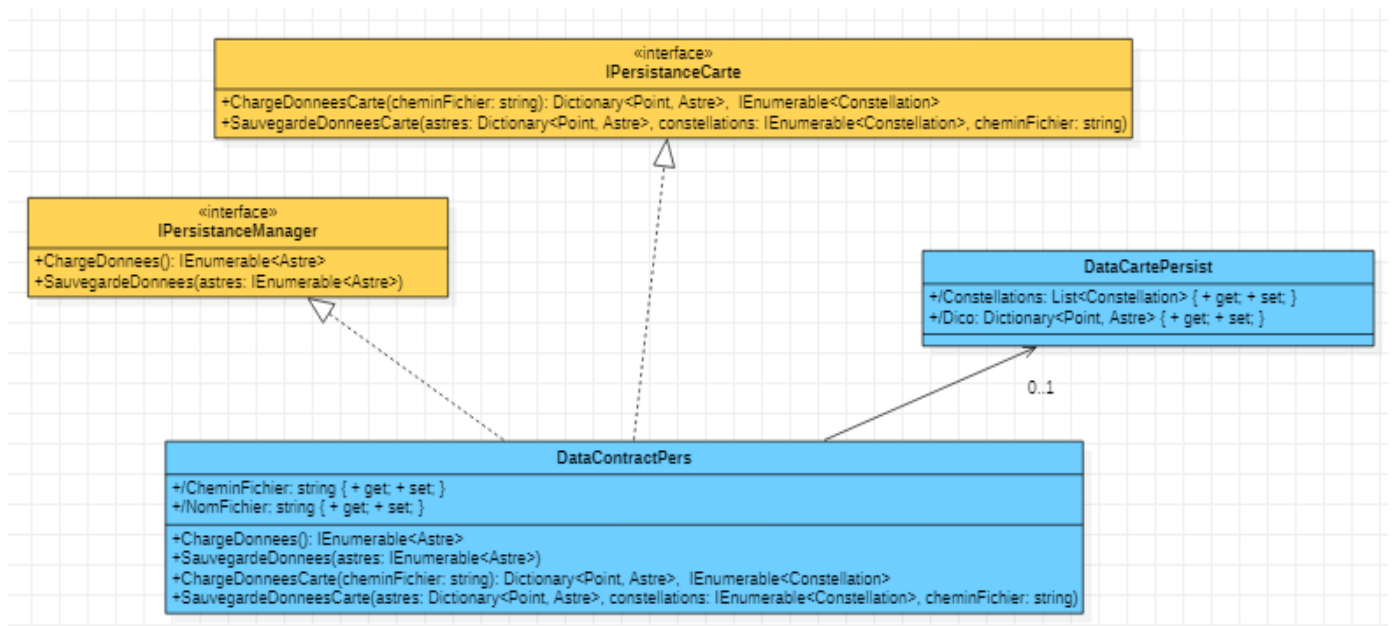
Il est bon de noter que le Manager possède deux autres méthodes : une de chargement et une de sauvegarde de la carte cette fois-ci. Il s'occupe ainsi de déléguer l'action aux méthodes de chargement et de sauvegarde de la Carte, précédemment décrites. Il ajoute aussi certaines données dans sa liste d'astres, afin que les astres chargés apparaissent également dans la liste du menu latéral.

De plus, les méthodes de chargement et de sauvegarde de la carte nécessiteront un chemin en paramètre, ce sera l'endroit où se trouve le fichier à charger / dans lequel les données seront sauvegardées. Notre Carte doit pouvoir être chargée n'importe où en fonction des choix de l'utilisateur, ces chemins seront donc déterminés dans notre *Appli*, la partie visuelle de notre projet.

Nous allons donc expliciter les divers modes de persistance qui seront utilisés par ces *IPersistence*. Nous avons déjà décrits le *Stub* qui est un mode de persistance plus ou moins fictif, nous allons maintenant décrire celui du *DataContract*.

3.4.3 Persistance en DataContract To XML

Ce mode de persistance permet de sérialiser en XML les données. Il est pratique si aucune forme n'est demandé dans la façon dont seront enregistrées les données dans le fichier.



Ce mode de persistance implémente donc *IPersistenceCarte* et *IPersistenceManager*, car il est utilisé dans la sérialisation de toutes les données de notre application, de celle de notre Carte et de notre Manager.

Cette classe implémente donc les méthodes correspondantes de chargement et de sauvegarde. Afin de sérialiser les données, il a fallu décorer les propriétés que nous souhaitons enregistrer (dans le cas du Manager, les propriété de la classe *Astre*, *Etoile* et *Planete*). Les méthodes s'occupent ensuite de charger / sauvegarder les données de manière autonome.

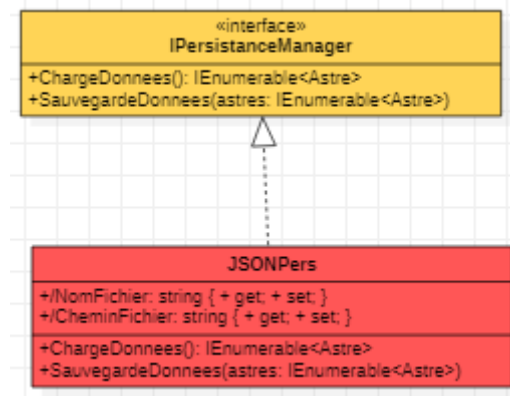
Cependant, la démarche est plus compliquée dans le cas de la Carte.

En effet, cette dernière ne possède non pas une seule collection, mais deux collections à sérialiser. Cependant, des références entre certains éléments doivent être préservées, on ne peut donc pas simplement sérialiser les données comme pour le Manager.

Il nous faut donc créer une classe *DataCartePersist*. Elle ne contient aucune méthode, seulement deux propriétés qui seront sérialisées. Ainsi, on ne va non pas sérialiser un dictionnaire de points et d'astres et une liste de constellation, mais on va sérialiser un *DataCartePersist* contenant un dictionnaire de points et d'astres et une liste de constellation.

En dé-sérialisant un *DataCartePersist*, on dé-sérialisera une collection de constellations, et notre dictionnaire. Voilà donc l'utilité de cette classe.

3.4.4 Persistance en JSON



La persistance en JSON utilise le *Nuget Newtonsoft* pour fonctionner. Elle fonctionne de la même manière que le *DataContract* pour le Manager, à savoir, elle implémente `IPersistenceManager` et ses méthodes. La méthode de sauvegarde prend en paramètre une collection d'astres à sérialiser et la méthode de chargement retourne une collection d'astres.

La persistance en JSON n'a pas été implémentée par la Carte, elle sert seulement à sérialiser à charger et sauvegarder les données du Manager.

3.5 Diagramme de classes sur les parties ajoutées

Dans notre application, nous avons un *master-detail* des plus classique, avec en plus certains systèmes de tris.

Mais nous avons en plus incorporé une partie « éditeur » dans laquelle l'utilisateur peut créer les cartes qu'il imagine, ses propres étoiles et planètes. Il peut les placer sur la carte, les déplacer, les relier (seulement les étoiles), les effacer... Et il peut enregistrer ses cartes pour les charger plus tard.

Cette Partie « Carte » est plutôt bien définie dans notre *Modele*. Nous avons une classe à son nom, qui comporte des méthodes concernant toutes ces actions pour relier, effacer, modifier, créer, déplacer...

Nous n'allons pas la décrire ici car nous l'avons déjà fait dans la partie 2.4.4 *Le Manager et la Carte* (pages 18-19).

Or cette Carte pour fonctionner a nécessité une implémentation particulière. En effet, elle est composée de points et de segments. Il nous a donc fallu créer un espace de nom *Geometrie*, dans lequel nous avons créé la classe *Point*, et la classe *Segment*. Nous avons utilisé ces points dans notre dictionnaire de points et d'astres, ce qui permet de faciliter la récupération de l'astre sur la carte à partir du point cliqué, cela simplifie donc grandement la gestion des données dans notre *Appli*. Nous avons également créé la classe *Constellation*, qui utilise elle aussi les points et segments.

Ces classes *Point*, *Segment* et *Constellation* ont été tous trois détaillé dans la partie 2.4.3 *Les constellations* (pages 16-17).

La persistance de cette Carte a été expliquée dans la partie juste au-dessus, très précisément

3.4.3 Persistance en DataContract To XML (page 37 essentiellement).

Nous avons également un système de favoris, qui permet de sauvegarder certains astres afin de les retrouver plus facilement. Cela a été expliqué rapidement dans la partie 2.4.1 *Le cœur de l'application : les Astres* (début page 15).

3.6 Vidéo promotionnelle

Cette dernière est trouvable sur la forge.

Note : tous les documents présentés ici sont trouvables sur la forge (dossier */trunk/documentation/*). Les diagrammes sont présents sous forme de fichiers *StarUML* (diagramme de cas d'utilisation, diagrammes de séquences, diagrammes de classes, diagrammes de paquetage). Certains d'entre eux sont disponibles sous forme de dessin SVG. Enfin la vidéo promotionnelle peut également être trouvée à cet emplacement.

Cela peut être utile car certains diagrammes sont assez gros, et peuvent être difficilement lisibles sur ce document.

Voici l'URL du dépôt : <https://forge.clermont-universite.fr/svn/stellar>