Description écrite de l’architecture :

Diagramme de Classes :

Architecture :

Manager:

Manager est un peu le point central dans notre code : Il permet de centraliser les fonctions les plus importantes et utiles et surtout il nous permet de faire un lien entre le code en c# et le XAML (dans notre cas).

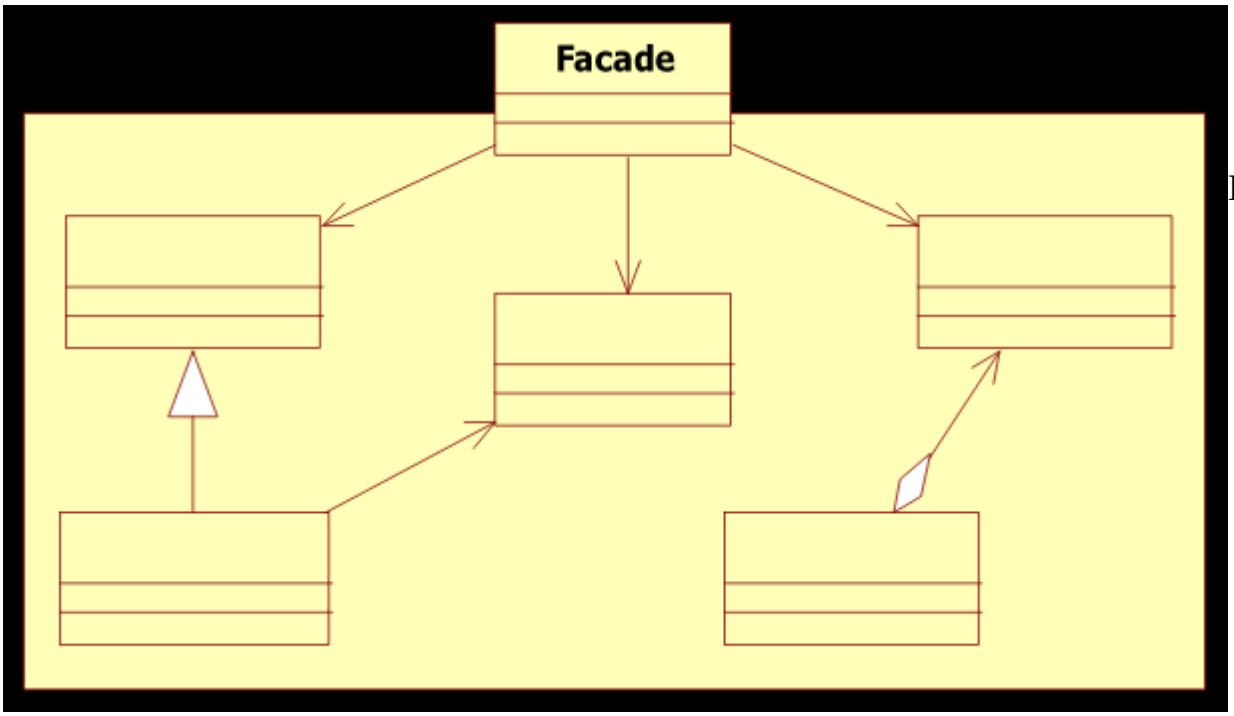
Pourquoi ?

Manager est ce qu'on appelle une "Façade"; une Façade est un intermédiaire entre un élément externe et un sous-système de classes qui propose une vue simple de ce sous-système car elle en cache la complexité pour seulement en montrer la façade (d’où son nom). Ainsi, les fonctionnalités sont limitées et donc recentrées sur les besoins du système externe et seulement sur le nécessaire. Depuis la façade on peut réaliser tout ce qui est normalement faisable par les classes du sous système. La Façade permet donc une indépendance du sous système en permettant au sous-système d'évoluer sans qu’elle ne change (à moins de changements majeurs bien sûr (ex : ajout d'une nouvelle bibliothèque de classes)). La façade est le point d'accès principal au sous système, ainsi, l'indépendance de celui-ci est permise car il ne dépend en rien du manager.

Dans notre cas, Manager permet une simplification du sous-système du côté de l’utilisateur, mais surtout un lien entre eux qui permet des interactions dans les deux sens. On garde ainsi l'indépendance du code vis à vis de la persistance car sans ça, on doit changer dans chaque classes plusieurs choses pour adapter le code (pour peu qu'on en ai énormément cela devient très lourd et long). Notre sous-système perdrait alors en ré-utilisabilité car il ne serait adapté qu'a ce changement et non à des possibles autres solutions.

Manager permet donc de lancer les processus de chargement de données et de sauvegarde, en gardant un impact très minime sur le reste du code (ajout de [DataContract] et [DataMember]). C'est lui qui dit à la Stratégie ce qu'on lui demande pour qu’elle puisse ensuite prendre le relais et appliquer la demande (on appelle cela une délégation).

Par exemple sur l’image d’en dessous on voit que la façade est en dehors du sous-système. Ce qui montre que la façade est le point d’entré de celui-ci mais qu’elle reste quand même en dehors du sous-système.



La

Stratégie :

Qu'est-ce qu'une Stratégie ? Dit simplement, c'est une manière de faire des choses différemment en obtenant des résultats similaires/homogènes. En d'autres termes ; si on prend une classe avec un comportement spécifique et qui le traite de différentes façons, on peut alors la décomposer en plusieurs classes séparées ; l'ensemble est ce que l'on appelle une stratégie. Ainsi, la classe originale (qu'on appelle contexte) doit garder une référence vers la classe abstraite de la Stratégie (dans notre cas une injection de dépendance par constructeur ayant pour paramètre un élément de type de l'interface IPersistanceManager). Pour pouvoir utiliser des stratégies concrètes et interchangeables qui implémentent la classe abstraite de la stratégie, le contexte délègue alors à la stratégie la réalisation de la fonction de la classe concrète choisie.

Dans notre cas : la Stratégie provient de l'interface IPersistanceManager (c'est le contrat entre la stratégie et la façade) et des classes qui l'implémentent.

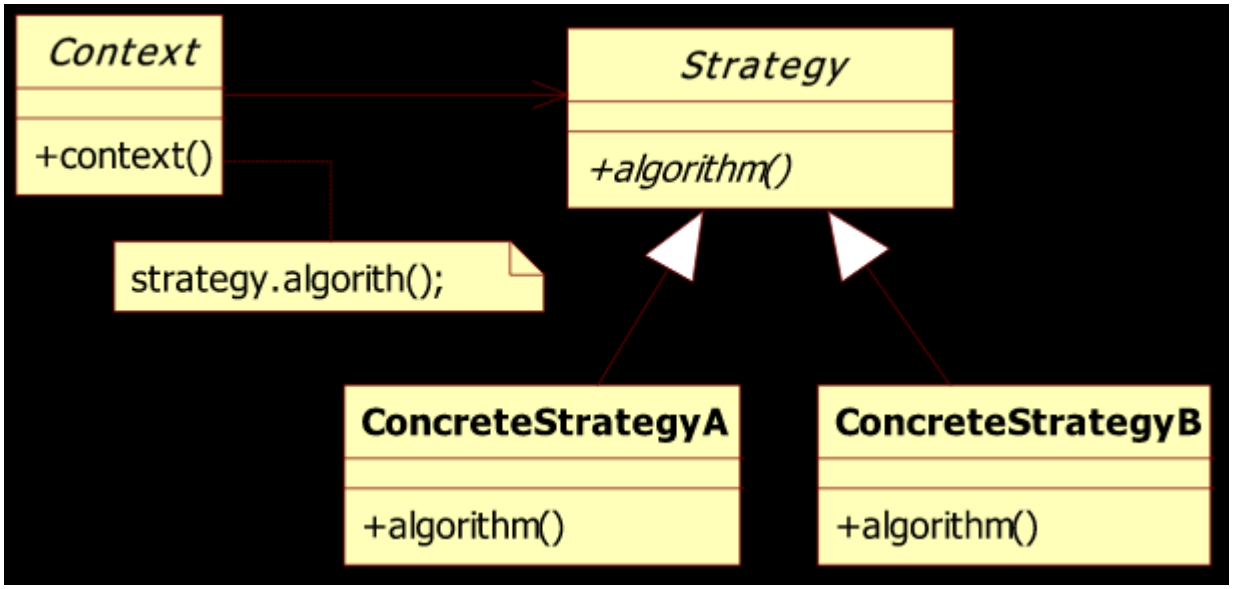
Soit :

(StratégiesConcrètes)
-Stub
-DataContractPers
-DataContractPersJSON

On obtient ainsi une extensibilité de nos stratégies concrètes ; on peut en ajouter sans qu'aucuns problèmes n'apparaissent (tant qu'elles fonctionnent).

L'objectif de cette Stratégie est de nous permettre d'implémenter de la persistance de plusieurs manières différentes (cf au dessus) tout en gardant l'indépendance de notre Manager. Celui-ci n'est directement lié à aucune des classes concrètes de la stratégie et n'a pas besoin de changer pour chaque stratégie concrète (ce qui s'explique par l'interchangeabilité des classes concrètes de notre stratégie). La stratégie de persistance nous permet donc de sauvegarder dans un fichier ou charger des données depuis un fichier.

Le reste des classes n'a pas de forme particulièrement spécifique à un patron de conception donc nous en parlerons dans la partie suivante.



On voit bien sur cette image la représentation de ce qu'on a dit plus tôt : Un contexte qui possède un algorithme particulier. Tandis que cet algorithme appelle dans la stratégie, la méthode qu'il souhaite parmi celles disponibles pour réaliser ce qu'il veut accomplir.

Description des choix inter-classes :

Classe Item :

On possède dans notre modèle plusieurs énumérations (4 au total : Certification, Rarity, Color, et Specification).

On précise qu'on utilise des énumérations pour simplifier le code et le raccourcir mais surtout qu'on le fait car les données sont connues et finies (imposées par le jeu).

-Certification liste les certifications qui sont disponibles dans Rocket League.
-Rarity permet lui de lister les raretés des différents items (exemple de Common qui veut dire Commun).
-Color est la liste des couleurs disponibles (Malheureusement pas de DarkSalmon).
-Specification permet de savoir ce qu'est exactement l'Item (exemple des antennes (Antennas) ou des roues (Wheels)…).

Ces 4 énumérations sont utilisées pour créer des Item qui prennent en paramètres :

-Type (un élément de Spécification converti avec ToString pour avoir sa valeur string et non sa valeur short) On l'utilise car il y a plusieurs types d'items différents, ainsi on doit pouvoir les différencier par ce critère. / On veut pouvoir trier par Type.
-Rarity (un élément de Rarity converti avec ToString pour avoir sa valeur string et non sa valeur short) Idem que pour type, il existe plusieurs raretés. / On veut pouvoir trier par rareté.
-Color (un élément de Color converti avec ToString pour avoir sa valeur string et non sa valeur short). On l'utilise car un Item peut avoir une couleur et qu'il y a une quantité limité disponible de couleurs à celles dans l'énumération Color. / On veut pouvoir trier par couleur.
-Certification (un élément de Certification converti avec ToString pour avoir sa valeur string et non sa valeur short). On l'utilise car un Item peut avoir une certification (ou non).
-Serie (un string correspondant au groupement de l'Item (c'est à dire des Items liés)). / On veut pouvoir trier par Serie.
-RealeaseDate (un DateTime correspondant à la date de sortie de l'Item).
-IsTradable (un bool qui permet de savoir si jamais l'Item est échangeable (certains ne le sont pas)).
-IsCertifiable (un bool qui permet de savoir si un Item peut avoir une Certification ou non (Il peut y avoir des Items non Certifiables et sans Certification MAIS il peut aussi y avoir des Items Certifiables sans Certification)).
-HasBluePrint (un bool qui permet de savoir si un Item possède un BluePrint ou non).
-Price (un int qui donne le prix de l'Item).

Item est notre classe la plus "basse" parmi les classes que l'on a créé car on avait besoin d'une classe pour rassembler les différentes caractéristiques des objets de Rocket League (cf ci-dessus).

Classe Set :

On possède aussi la classe Set qui possède :

-NumSet (un int qui représente son Numéro (pour le différencier des autres Set)). Au cas ou des Sets soient identiques.
-Name (un string qui a le nom du Set) pour que le set soit différentiable des autres.
-ReadOnlyDictionary<Item, int> SetItems (un dictionnaire en lecture seule (pour éviter que l'utilisateur casse tout avec des bêtises) que l'on restreint à l'appel des fonctions autorisées). On le crée à partir de setItems (un Dictionary<Item,int>) qu'on utilise dans le constructeur sous la forme suivante :
"SetItems = new ReadOnlyDictionary<Item, int>(setItems);"
L'encapsulation permet une sécurité quant au fonctionnement à toute épreuve (ou presque) du code).
Encapsuler le dictionnaire empêche l'utilisateur de l'utiliser, car celui-ci est en read only. « add » pourrait par exemple casser les Set en ajoutant plus d'un Item de chaque type (Or, clear, add … ne sont pas implémentés par cette classe. On n'a pas de problème avec ça et l'utilisateur est donc limité dans ses actions sur le système). On utilise un dictionnaire pour avoir en valeur une correspondance avec l'énumération Spécification (un set ne peut pas avoir deux fois le même type d'item).

Set possède des méthodes bien à lui :

-SetAdd est une méthode particulière qu'on peut apparenter à la méthode Add des collections mais en bien plus poussée dans le cas présent :
Elle prend en paramètres :
-Un Item item (soit l'item à ajouter).
-Un bool Change (permettant de savoir, si un Item avec le même type existe déjà, si on veut écraser ou non).

Elle vérifie si le dictionnaire vaut null, au quel cas on stoppe la fonction. Si le nombre d'éléments vaut TAILLEMAX (variable statique qui vaut 14 (soit le nombre de Spécifications)) et que Change vaut false alors on ne peut rien faire et on stoppe la fonction.

Sinon, en fonction des différents cas elle agit différemment :

-Si le dictionnaire SetItems ne possède pas d'Item du type de celui qu'on veut ajouter, alors on ajoute forcément l'Item.
-Si il y a un Item avec le même type et que Change == false, alors on ne fait rien et on stoppe la fonction (dans le cas ou l'utilisateur fait une action non voulue).
-Enfin, si il y a un Item qui existe déjà avec le type de l'Item voulu et que Change == true, alors on cherche avec un « foreach » quel est l'item, puis on donne à un int « j » sa Value. On retire l'ancien Item et on ajoute le nouveau avec pour Value notre « j » (soit la Value de l'ancien Item) ; de manière grossière, on le remplace.

On utilise cette méthode pour gérer les ajouts, dans le cas ou un ajout conduirait à un doublon de type, pour que un set ne contiennent jamais plus que un type d'Item à la fois, elle permet de gérer automatiquement selon ce que met l'utilisateur, les différents cas de figures énoncés plus haut et on à donc pensé qu'elle était adaptée à la situation.

-SetNumSet permet de définir le numéro d'un Set (après l'avoir recherché grâce à LINQ) en fonction de si celui-ci est déjà attribué à un autre Set (que l'on trouve dans une liste). On trie ensuite la liste. On utilise cette méthode pour éviter un doublon lors de la modification du numéro d'un set.
On veut éviter le cas du doublon pour pouvoir se retrouver dans les Set et avoir une unicité certaine.

-DeleteItemFromSet (prend en paramètre un Item et l'enlève du set si il y est). La méthode est utilisée pour retirer un Item d'un set (elle est utilisée par le manager dans sa fonction DeleteItem).

Classe CompteGuest et Compte:

CompteGuest est une classe très simple faite spécialement pour le mode « guest » de l'application. Il possède un Login (un string pour le pseudo de l'utilisateur) donné par le constructeur ("Guest") puis l'on crée aussi un Set SetActif (avec pour nom SetActif). Son ToString renvoie que le compte disparaîtra car CompteGuest n'est pas fait pour durer (ce qui et la raison de pourquoi cette classe est si courte (et porte ce nom)).

On à décidé de créer cette classe car pour un Guest (un invité) sur une application ou même un site web, il n'y a jamais trop de fonctionnalités.
Pourquoi ?

Car un CompteGuest n'est pas enregistré dans la base de données, il est détruit une fois qu'on ne l'utilise plus à l'instar des Compte classiques qu'on enregistre. C'est donc car il est éphémère que CompteGuest est plutôt vide ...

Compte hérite de CompteGuest donc de sa propriété Login (modifiable dans Compte et non prédéfini) mais ajoute aussi :

Pour éviter de créer 2 classes distinctes alors qu'elle ont toutes deux des points communs, on a implémenter un héritage qui permet de raccourcir le code et d'éviter l'effet "déja vu" sur la classe Compte.

-Password (un string qui contient le mot de passe du compte). Nécessaire pour s'identifier ensuite (tout comme le Login de Compte).

-Email (un string qui contient l'adresse Email du compte). Si on a besoin de contacter l'utilisateur en cas de problème ou suite à d'autres actions.

-ReadOnlyCollection<Set> Sets (Même idée que pour SetItems de Set mais ici pour ranger des sets et encapsuler la collection (qui est une liste définie par Sets = new ReadOnlyCollection<Set>(sets); (sets qui est de type : List<Set> sets))). Similaire à notre ReadOnlyDictionnary, on utilise ici une ReadOnlyCollection pour encapsuler une collection

(ici une liste comme précisé) avec toujours pour but d’empêcher l'utilisateur de casser le code en faisant des choses qu'on ne voudrait pas (clear, add … ne sont pas implémentée par cette classe donc on n'a pas de problème ici aussi). Le but comme énoncé est de limiter l'impact de l'utilisateur sur le code.

Compte ajoute :

La méthode CompteAdd qui prend un Set et qui si celui-ci n'est pas dans la liste des sets du compte est ajouté. Sinon on renvoie la fonction. On veut que chaque Set n’existe qu'une fois dans la liste de Compte.

-La méthode CompteSetNumSet permet de faire appel à SetNumSet dans Compte en utilisant Sets comme liste de Set. On utilise l'appel à la fonction de SetNumSet pour éviter d'avoir un NumSet avec un setter en public et donc la possibilité que l'utilisateur fasse des siennes.

-DeleteSetFromCompte (prend en paramètre un Compte et un Set puis enlève ce dernier si celui-ci existe déjà dans le Compte). La méthode est utilisée pour retirer un Set d'un compte (elle est utilisée par le manager pour sa fonction DeleteSet).

Compte représente l'utilisateur, c'est lui qui rassemble ses informations personnelles ainsi que ses créations (ses sets qui sont eux mêmes remplis d'Items), on l'utilise donc pour permettre la création d'un compte et d'effectuer des modifications ainsi que de supprimer ce dernier.

Manager :

Manager possède en attributs :

-AllItems (une ReadOnlyObservableCollection<Item> pour pouvoir stocker des données qui vont agir directement sur le XAML car la modification ou suppression de l'une de ces données sera apparente sur le XAML). Elle possède une ObservableCollection<Item> pour qu'on puisse interagir avec elle qui stocke tous les items créés (utilisés ou non).

-AllSets (une ReadOnlyObservableCollection<Set> pour pouvoir stocker des données qui vont agir directement sur le XAML car la modification ou suppression de l'une de ces données sera apparente sur le XAML). Elle possède une ObservableCollection<Item> pour qu'on puisse interagir avec elle qui stocke tous les sets créés.

-AllComptes (une ReadOnlyObservableCollection<Compte> pour pouvoir stocker des données qui vont agir directement sur le XAML car la modification ou suppression de l'une de ces données sera apparente sur le XAML). Elle possède une ObservableCollection<Item> pour qu'on puisse interagir avec elle qui stocke tous les comptes existants

-Persistance (de type IPersistanceManager, cet attribut est celui qui permet la sauvegarde et le chargement des données)

Manager possède certaines fonctions comme :

-DeleteCompte (prend un Compte et détruit les références qu'il possède).
-DeleteSet (prend un Set, un Compte, un bool Everything et détruit les références que le Set possède (si Everything vaut true, on le détruit partout, sinon on le détruit dans le compte donné en paramètre)).
-DeleteItem (prend un Item et détruit les références que l'Item possède).

Ces méthodes servent à "supprimer" ce qu'ils prennent car en supprimant les références qu'ils ont, le garbage collector viendra les détruire). On veut les utiliser si on ne veut plus de tel ou tel chose par exemple.

-CreateCompte / CreateSet / CreateItem (On comprends ce qu'elles font.) Elles sont des fonctions simples qui servent juste à simplifier la vie pour le côté utile.
-ChargeDonnées (charge les données grâce à l'attribut Persistance de type IPersistanceManager depuis la stratégie).
-SauvegardeDonnées (sauvegarde les données grâce à l'attribut Persistance de type IPersistanceManager depuis la stratégie).

Mais on à aussi des fonctions (pas encore implémentée ni écrites) :

-ExportSet qui permettra de partager ses créations à d'autres utilisateurs
-ImportSet qui permettra de recevoir une création de quelqu'un d'autre

(Qu'on implémentera si on à le temps de le faire)

-(Peut-être d'autres)

Le Manager possédera aussi des fonctions pour la passerelle entre les classes et le XAML .

Stratégie :

Notre stratégie contient plusieurs classes dans nos bibliothèques de classes StubLib et DataContractPersistance.

Dans StubLib :

On trouve la classe Stub (une classe de persistance un peu en carton) car :

Elle possède des méthodes qu’elle tient de l’interface IPersistanceManager (SauvegarderDonnées et ChargeDonnées).

Et d’autres méthodes propres à elle :

-ChargeItems
-ChargeSets
-ChargeComptes

Qui, respectivement :

Crée des Item, les mets dans une liste et la renvoie.
Crée des Item, des Set, met des Item dans les Set et les Set dans une liste de Set et la renvoie.
Crée des Item, des Set, des Compte, met des Item dans les Set et des Set dans les Compte, met les Compte dans la liste de Compte et la renvoie.

Ces méthodes sont appelées par ChargeDonnées qui va renvoyer ces listes à l’appelant.

Pourquoi le Stub est une classe « en carton » ?

- Elle ne charge que les mêmes données, ne sauvegarde rien.

Dans DataContractPersistance :

On trouve plusieurs classes : DataContractPers, DataContractPersJSON et DataToPersist.

DataToPersist : Une classe qui possède en attribut une liste d’Item, une liste de Set et une liste de Compte. Elle sert à éviter de compliquer le code pour la persistance car on l’utilise pour persister nos 3 listes.

DataContractPers : Elle possède 3 attributs normaux et 2 attributs calculés.

-RelativePath qui est le chemin relatif du fichier.
-FileName qui est le nom du fichier.
-Serializer qui nous permet de sérialiser le fichier en gardant les références aux objets (éviter les doublons) au format XML (de type DataContractSerializer).
-FilePath qui est une propriété calculée permettant de remonter au dossier voulu depuis le dossier actuel.
-PersFile qui permet d’avoir l’emplacement et le Nom du Fichier pour pouvoir l’éditer ou le lire. Elle implémente les méthodes de l’interface IPersistanceManager soit :

- ChargeDonnées vérifie que le dossier existe, sinon elle le créé. On créé un DataToPersist data auquel on donnera les valeurs de ce qui est lu avec Serializer après avoir ouvert un chemin dans le fichier. Puis on retourne les listes de data.

-SauvegarderDonnées prend en argument un IEnumerable d'Item, de Set et de Compte. Vérifie que le dossier existe, sinon elle le créé. On créé un DataToPersist data auquel on donne les valeurs des IEnumerables passés en arguments. On créé des settings d'indentation. Puis on utilise un TextWriter dans le fichier grâce à PersFile qui nous permet d'utiliser un XmlWriter writer pour pouvoir modifier/créer un fichier XML grâce à writer et data.

DataContractPersJSON : dérive de DataContractPers et possède donc ses attributs et méthodes. Elle modifie les attributs non-calculés pour les adapter au format JSON dans son constructeur (Serializer devient un DataContractJsonSerialier pour palier à ça).

Elle modifie la méthode de sauvegarde pour qu'elle ne prenne plus le format XML et qu'elle lui soit ainsi adapté : on à toujours data notre DataToPersist. Mais on utilise un simple canal (writer) qui créé un fichier en fonction de PersFile tandis que Serializer permet l'écriture en fonction de writer et data.

On utilise ces fonctions et leur contenu pour pouvoir rendre nos Item, Set et Compte persistants afin de les réutiliser.

Diagramme de Paquetages :

Dans notre Appli, on possède une bibliothèque de Classes (où l'on trouve nos classes) dont dépendent nos 5 Tests :

-Test_Items nous permet de vérifier si un Item se forme comme on le souhaite.
-Test_Set permet de vérifier si un Set se forme correctement, puis si il peut contenir des Item et si ses méthodes fonctionnent correctement.
-Test_Compte permet de vérifier ce qu'il se passe pour un Compte et un CompteGuest et si les méthodes de Compte fonctionnent correctement.
-Test_PersXML vérifie qu'on sauvegarde/charge correctement des données dans un fichier XML.
-Test_PersJSON vérifie qu'on sauvegarde/charge correctement des données dans un fichier JSON.

Diagrammes de Séquences :

Le premier diagramme de séquence représente la création de 2 items ; item1 et item2 (on pourrait en rajouter), leur ajout, la création d'un Set set1, dans set1 avec la fonction SetAdd (expliquée plus haut). Ensuite on crée un Compte compte1 puis on ajoute set1 dans le Sets de compte1 avec CompteAdd (expliquée plus haut). Et enfin on appelle le ToString de compte1 et de set1 qui donnent respectivement les informations du compte ainsi que celles du set qui contiennent celles des items.

Le second diagramme de séquence représente le chargement depuis un stub appelé par le Manager. On appelle, après la création du Manager, la fonction ChargeDonnée de celui-ci qui appelle la fonction éponyme du stub (dépendance choisie). Elle créé des Item, des Set et des Comptes puis renvoie une liste de ceux-ci. On récupère le contenu et l’ajoute aux listes du Manager.