**Description de l’architecture de notre application**

**Présentation des nuGets.**

Tout d’abord pour la réalisation de notre application, nous avons utilisé 3 ressources différentes.  
Le package nuGet AutoFixture qui nous permet de nous concentrer sur nos tests, il nous permet d’écrire moins de code pour nos tests.  
Le second nuGet est NETStandard.Library, ce nuGet permet d’uniformiser tout notre code .NET .   
Il va nous permettre de définir un ensemble d’uniforme d’interface de programmation de bibliothèques de classes. Pour toutes les implémentations de .NET on aura des bibliothèques de portables de classes utilisables. Grâce à l’ensemble de nos API (Application Programming Interface ou interface de programmation).

Le dernier nuGet utilisé est Newtonsoft.Json. Il est utile pour notre partie persistance. Il nous permet de sérialiser et désérialiser à l’aide de C#.NET. Grâce à sa méthode JsonSerializer.Serialize, on peut créer JSON sous forme d’une chaîne de caractère. Et avec la méthode JsonSerialiser.Deserialize on peut désérialiser c’est-à-dire lire Json dans des objets .NET. Donc il nous permet d’écrire en Json et de lire Json.

Notre partie « Modèle » utilise les nuGets AutoFixture et Newtonsoft.Json et Newtonsoft.Json.

**Explication des liaisons entre notre partie Modèle et notre Partie Vue.**

Quand on clique sur un bouton (dans notre partie vu) ça ouvre une fenêtre qui déclenche une méthode (dans le modèle) cette méthode, modifie le modèle qui permet de modifier la partie vue, par exemple un bouton modifier dans la partie compétences lorsqu’on clique il déclenche l’évènement buton\_click qui fait apparaitre une textbox dans laquelle on peut écrire quelque chose dans ce cas, qui nous permet de modifier le nom d’une ou plusieurs compétences.

Une partie important qui permet la liaison entre la partie vue et la partie modèle est la partie Binding. Ils permettent d’associer une propriété cible à une propriété source. Par exemple au début il faut définir un contexte, dans notre cas c’est notre classe Manager. Ensuite il faut réfléchir à 4 points,

Quelle est la source du binding ? Quelle est sa propriété ? Quelle est la cible du binding ? Et Quelle est sa propriété ?

Description d’un binding comme exemple : pour faire en sorte qu’une description corresponde à un héros. La source du binding est l’attribut description du héros, la propriété est la propriété text et la cible est un text-block. C’est un exemple. En effectuant un binding l’application va reconnaître quel héros possède quel description. Elle ne va pas associer la description du héros Hanzo au héros Ange par exemple. Chaque héros aura ses informations propres à lui.

**Patrons de conception :**

On utilise 2 patrons de conceptions dans notre application : Strategy et Façade

Exemple d’utilisation de notre patron de conception Strategy :

Nous avons instancié plusieurs constructeurs dans la classe Héros par exemple.

L’un a besoin d’un nom d’une image et sa classe et ensuite ca définit les autres options aléatoirement. (Les pourcentages des détails compétitifs sont définit aléatoirement entre 1 et 100 par exemple). La description du héros dans l’onglet biographie reprend « Le nom du héros » est une personnage du type « Sa classe ». C’est une phrase pré faite.

Le second constructeur consiste à tout donner en paramètre. Il est plus précis mais beaucoup plus complexe. Ce constructeur a un prénom, une image, une classe, une description, un média, les cosmétiques, les données compétitives, les compétences et les vidéos.

Le premier constructeur sert pour les visiteurs du site qui rajoute un héros, et le second constructeur sert pour l’application qui lorsqu’elle se lance charge tous les héros avec leurs informations complètes. Les deux constructeurs associent une couleur en fonction de la classe choisit, couleur bleue pour les DPS, couleur jaune pour les healers et rouge pour les tanks. L’application en fonction de ce qu’on lui demande va privilégier un constructeur plutôt que l’autre.

Exemple d’utilisation de notre patron de conception Façade :

Notre classe Manager correspond au patron de conception Façade, il nous sert d’interface qui permet d’utiliser les autres classes plus facilement. La classe Manager permet d’identifier les requêtes des utilisateurs afin de choisir la sous classe appropriés qui doit effectuer cette requête. La classe Manager permet de structurer notre application et rendre notre code plus lisible. Manager utilise les autres classes de la bibliothèque afin de leur ordonner d’effectuer les requêtes.

Le Manager gère les ajouts de différents éléments et les suppressions. C’est cette même classe qui permet de choisir le type de persistance, on peut facilement changer d’une persistance Json à une persistance factice Stub.

La sauvegarde de notre application s’effectue grâce à une méthode App\_closing qui lorsque l’application se ferme, toutes les modifications sont enregistrés dans le fichier BDD.Json.