Description des package

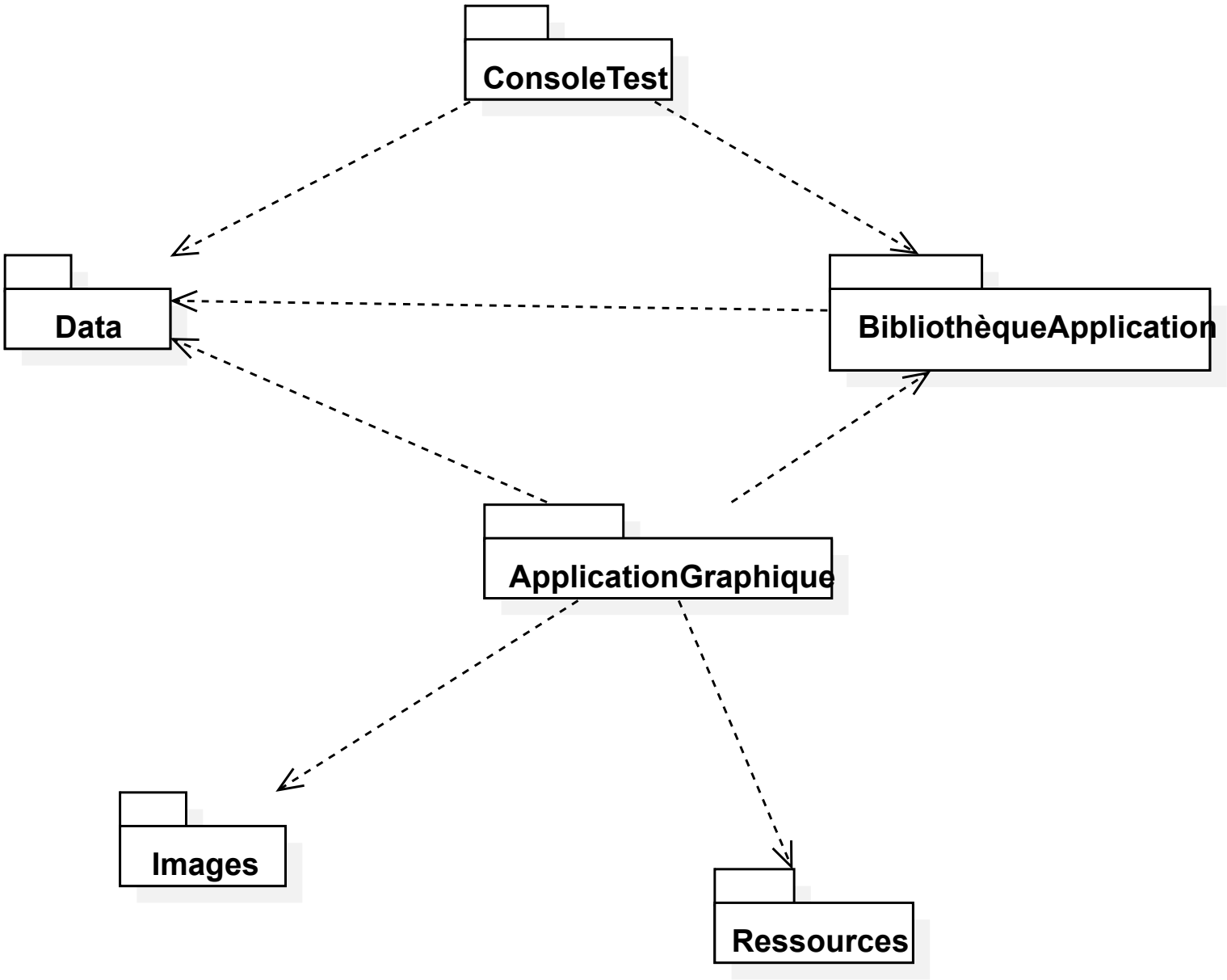
Le package nommé *BibliothèqueApplication* contient toute nos classes de notre application. C'est ici que se trouve tous les méthodes principales de notre application. Ce package dépend du package Data. Ce dernier contient notre classe Stub. Elle nous permet de gérer des données qui seront utilisées dans l'application tout comme dans les tests fonctionnels. Ce package ne dépend d'aucun autre package.

Le package Data sert à la persistance.

Le *ConsoleTest* est le package comportant tous nos tests. Il nous a fallu faire des tests pour vérifier si nos méthodes fonctionnaient correctement. C'est ici qu'ils se font. Il doit dépendre de Data pour pouvoir avoir les données pour réaliser les tests. Elle a aussi besoin de dépendre de la *BibliothèqueApplication* pour pouvoir utiliser les méthodes des différentes classes.

Pour finir *ApplicationGraphique* contient toute la partie graphique, en XAML, de l'application. Elle doit dépendre de Data pour pouvoir utiliser les données et les afficher dans les vues. Elle utilise aussi les méthodes des classes dans *BibliothèqueApplication*. Elle dépend aussi du package *Images* qui contient toutes les images servant à faire la partie graphique.

Le package *Ressources* est le package contenant toutes les images utilisées pour les jeux de notre application. *ApplicationGraphique* dépend des ressources.


ConsoleTest
Data
BibliothèqueApplication
ApplicationGraphique
Images
Ressources

Description de l'architecture

Les différentes classes de mon package *BibliothèqueApplication* dépendent les unes des autres. En effet la classe Jeu, qui est l'une des classes principales, dépend de la classe *InformationsJeu* parce qu'un jeu a besoin d'avoir des informations tel que le nom, la date de création ou encore un synopsis. Un jeu comporte aussi des théories et des visuels. Nous avons donc fait deux classes différentes nommées Théorie et Visuel. Cela permet de rendre le programme plus optimisé puisque c'est possible de modifier les théories et les visuels plus facilement.

Notre classe qui gère toutes les classes est notre classe Manager. C'est elle qui lie le modèle et la vue. C'est dans cette classe que nous utilisons nos propriétés pour le binding. Pour ne pas avoir à faire de la persistance sur le manager nous avons fait une classe StockApp qui permet d'instancier notre dictionnaire. Le Manager dépend alors de StockApp pour pouvoir manipuler notre dictionnaire de franchises. C'est comme cela que nous relions les franchises et les jeux. De plus pour pouvoir utiliser nos données, notre class Stub dépend de StockApp. Nous avons fait le choix de faire un Stub pour pouvoir tester nos méthodes avec des données qui ne seront pas finales.

Nos classes filles de Nommable ont toutes des noms et nous trouvions plus judicieux de faire des classes qui hériteraient de nommable. De plus nous faisons de l'héritage aussi entre les CreateurJeu, qui est la classe mère, Createur et Studio qui sont les classes filles. Il était plus simple de faire comme cela si nous voulions ajouter des informations sur les studios de production des jeux. En effet cela nous permet d'avoir une possible évolution pour notre application. Pour le moment nous ne voulons pas vraiment ajouter ce genre d'information mais il est possible de le faire à l'avenir.

Notre classe MainWindow, qui est la seule vue de la partie graphique utilise comme toutes les classes de notre partie graphique la propriété LeManager, de type Manager, dans App. Cette dernière est notre instance de manager utiliser dans toutes les classes pour les relier entre elles.

Pour la partie graphique nous avons décidé de faire seulement une seule vue. En effet nous avons remarqué que si nous faisons plusieurs vues cela dupliquerait du code. Une seule nous suffit. Nous utiliserons donc plusieurs UserControl et faire de la navigation entre eux pour rendre notre application fonctionnelle. Pour cela la class Navigator va nous permettre de faire la navigation entre les différents UserControl.

Description des classes

Dans notre diagramme de classe nous utilisons une classe abstraite nommée Nommable. Celle-ci nous permet de faire de l'héritage. En effet les classes *InformationsJeu, Théorie, CreateurJeu* et *Franchise* dérivent de la classe nommable. Celle-ci ne contient qu'un attribut *Nom* de type string.

La classe *CreateurJeu* est aussi une classe abstraite, mère de *Createur* et *Studio*. En effet, faire des classes séparées, rend le code plus réutilisable. Si nous voulions ajouter des informations sur le *Createur* et *Studio* il est possible de le faire. Nous avons fait ce choix pour laisser la possibilité de faire évoluer notre application.

La classe *InformationsJeu* est une classe qui a pour attribut dateCreation, limiteAge et un synopsis. Cette classe contient une liste de *Genres* et de *Plateformes*. La classe dépend donc des

enum *Genres* et *Plateformes*. Il est donc possible d'ajouter et de supprimer des genres ou des plateformes avec les classes *ajouterGenres(), ajouterPlateforme(), supprimerGenre(), supprimerPlateforme().* Ces méthodes prennent en paramètre soit un genre, de type *Genre*s, ou alors une plateforme, de type *Plateforme*s. La classe *Jeu* va dépendre aussi de la classe *InformationsJeu*. Les instances de la classe *InformationsJeu* existent seulement quand un *Jeu* existe. La *Vignette* correspond à l'image du jeu. Pour nous un jeu peut avoir beaucoup d'informations. Dans notre application nous voulions attribuer une liste d'images, théorie et musique en plus de différentes informations qui font qu'un jeu est un jeu dans notre application. Nous trouvions plus intéressant de partager ces informations en deux classes : *InformationsJeu* qui constitue les informations de basiques d'un jeu tel que le nom, le créateur, la limite d'âge ; et *Jeu* qui instancie des informations de jeu tout en ayant des listes de visuels, musiques et théories.

Nous avons dû faire les classe *EnumDescription* et *EnumExtension* pour pouvoir écrire les enum comme nous le voulions. Nous avons associé aux noms des enum un autre texte qui sera lui afficher sur la console et donc manipulé par l'utilisateur.

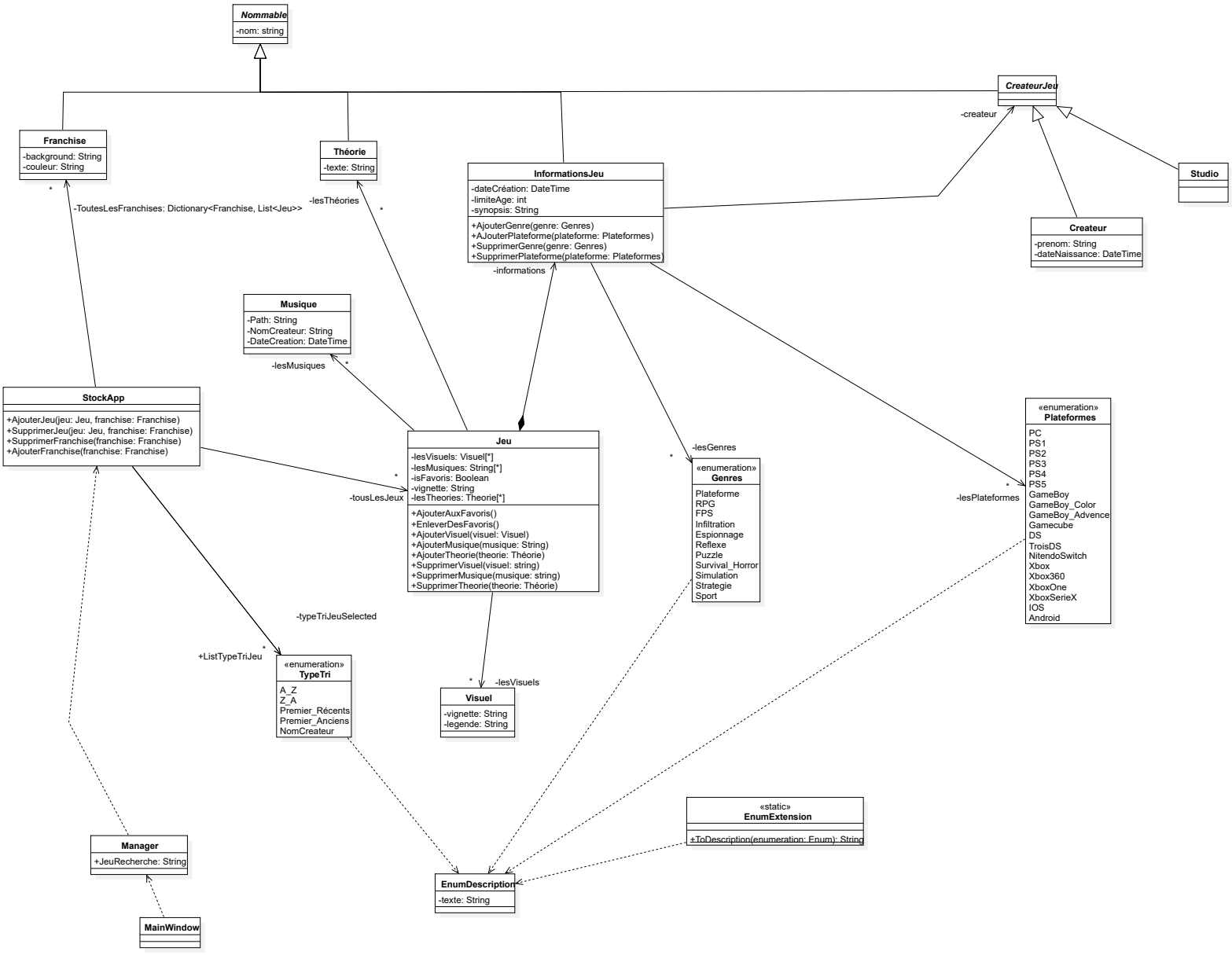
Le booléen IsFavoris permet de déterminer si le jeu est mis en favoris ou non, quand il est true le jeu est dans les favoris. Les méthodes qui rendent un jeu favori ou non sont *ajouterAuxFavoris()* et *enleverDesFavoris()* qui ne prenne rien en paramètre. Cette classe contient trois listes : une liste de visuels, de musique et de théorie. Il n'y a pas de nombre défini de ces éléments dans la liste. Pour faciliter la manipulation des éléments, nous avons préféré faire les classes *Théorie* et *Visuel*. La classe *Jeu* dépend des deux classes *Visuels* et *Théorie*. Nous avons des méthodes qui permettent d'ajouter des visuels (*ajouterVisuels()*), des musiques (*ajouterMusique())* et des théories (*ajouterTheorie())*. Ces méthodes prennent en paramètre l'élément à ajouter. Nous avons aussi des méthodes permettant de les supprimer : *supprimerVisuel()*, *supprimerMusique()*, *supprimerTheorie().* Ces méthodes, comme les méthodes d'ajout, prennent en paramètre l'élément à supprimer.

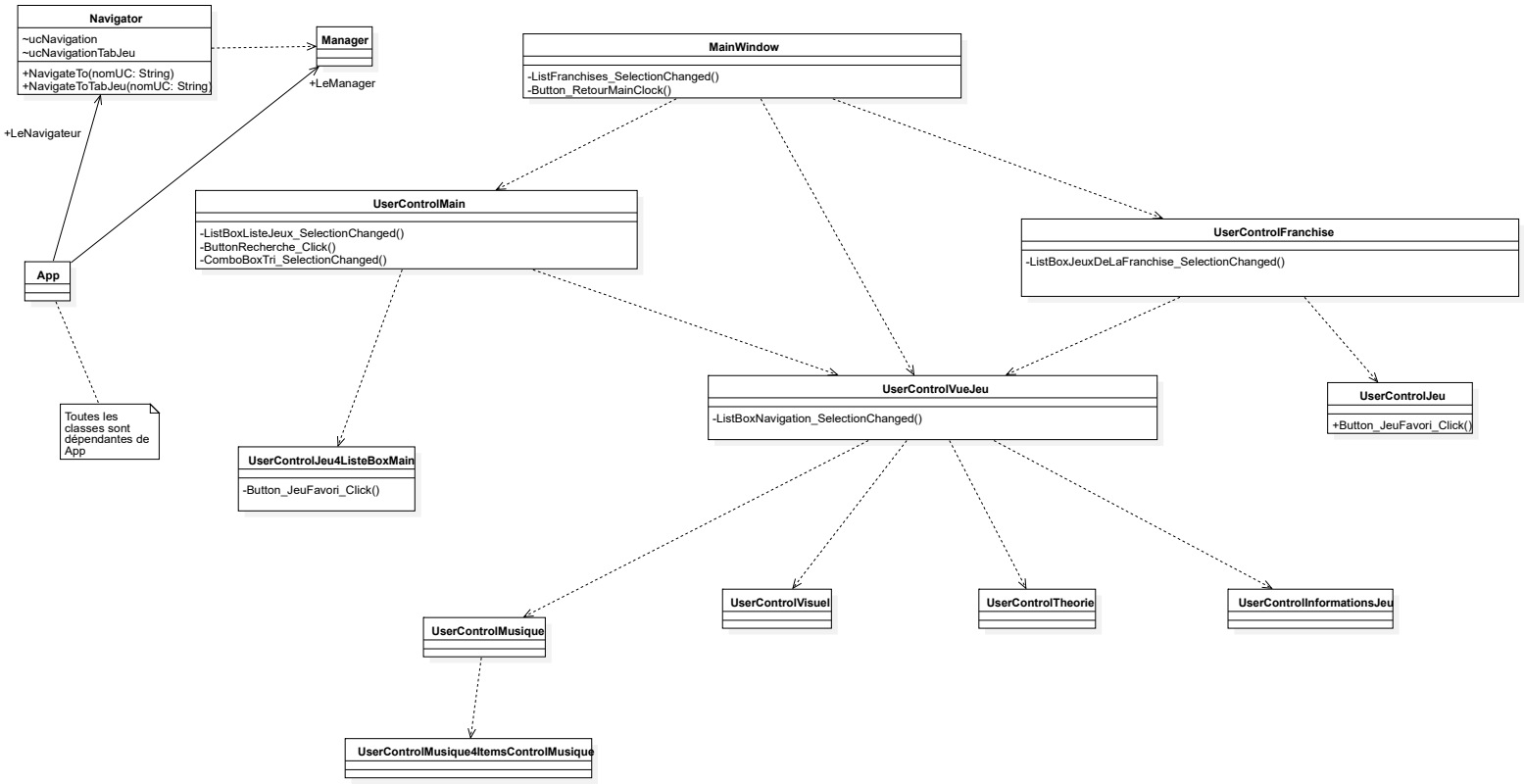
Nous avons aussi une classe *Franchise* qui a pour attribut un *background* de type string et une *couleur*. Ces attributs nous permettent de relier le code à la vue.

Notre application contient une classe *Manager* qui permet de gérer un dictionnaire de franchises. En effet nous avons décidé de faire un dictionnaire qui a pour clef les franchises et pour valeur une liste de jeux correspondant à la franchise. Cette classe contient une propriété calculée TousLesJeux permettant de faire un tri ou alors une recherche dans la liste tousLesJeux. Pour associer les différents types de tri nous avons fait un enum *TypeTri* contenant les différents types de tri que l'utilisateur pourra choisir sur la vue. Le Manager dépend alors de la classe StockApp car elle va stocker le dictionnaire. Elle contient la méthode *ajouterJeu()* vérifie si je le jeu n'est pas déjà existant avant de l'ajouter. Pour cela on va parcourir toutes les listes de jeux. S'il est déjà existant, on ne l'ajoute pas. Si ce n'est pas le cas on ajoute la franchise correspondante (appel de la méthode *ajouterFranchise())* et on ajoute le jeu dans la liste des jeux correspondant à la franchise. Il est possible de supprimer un jeu dans une franchise grâce à la méthode supprimerJeu(). Elle ne supprime seulement le jeu mis en paramètre. Pour supprimer une franchise il faut utiliser la méthode supprimerFranchise() qui prend la franchise à supprimer. En supprimant la franchise, la liste des jeux associées à la franchise est elle aussi supprimée.

La classe Navigator permet la navigation dans notre fenêtre *MainWindow*. On peut passer de la « page » principale aux franchises, jeux, aux favoris… Séparer cette navigation dans une classe à part est plus simple d'utilisation car elle donne accès, notamment, au UserControl actuel ce qui permet d'en changer. Mais cela permet aussi de, dans le futur, pouvoir rajouter des UserControl

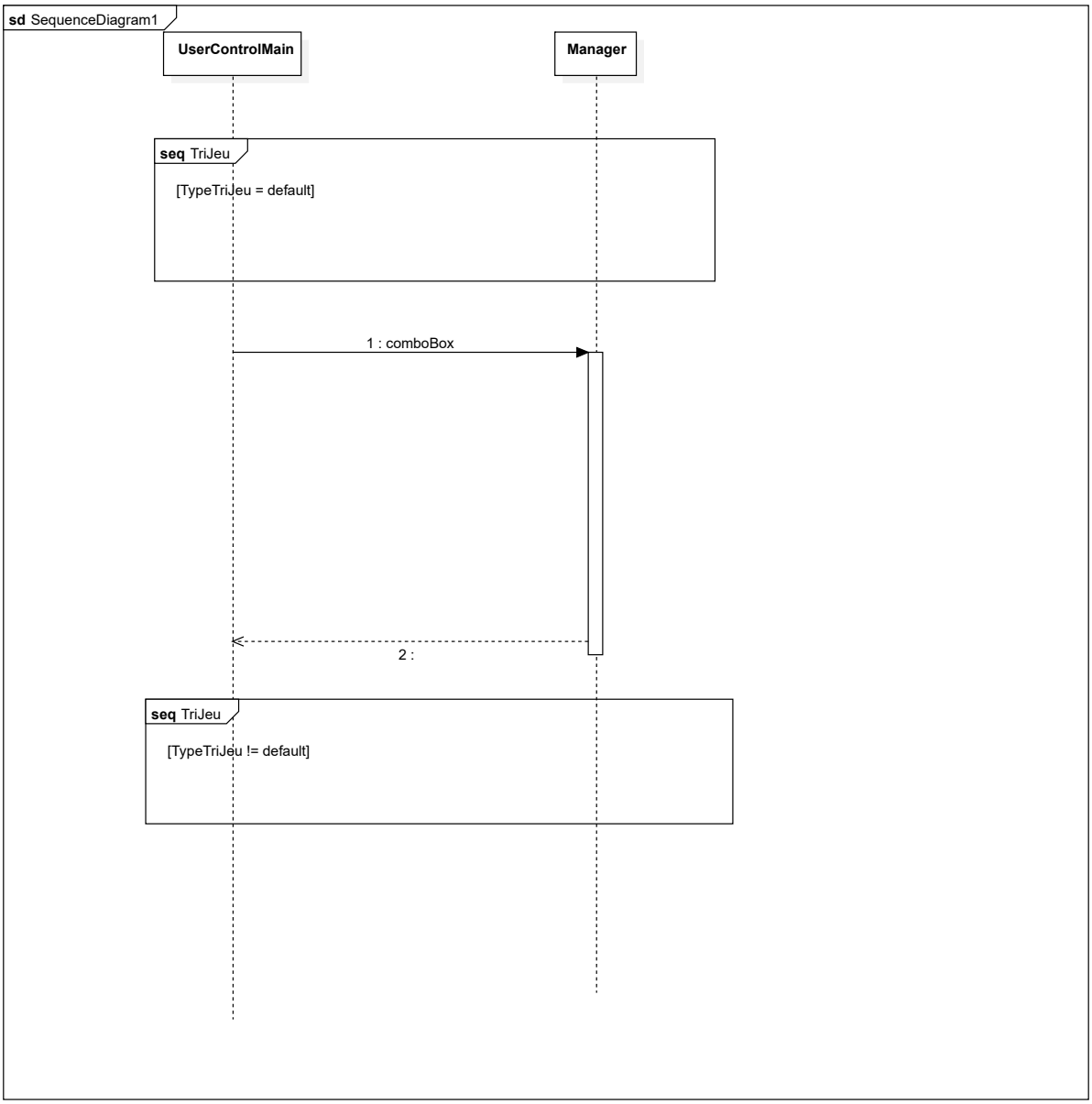
accessibles en ne modifiant qu'une seule classe rendant la maintenance et l'amélioration de l'application beaucoup plus simple.

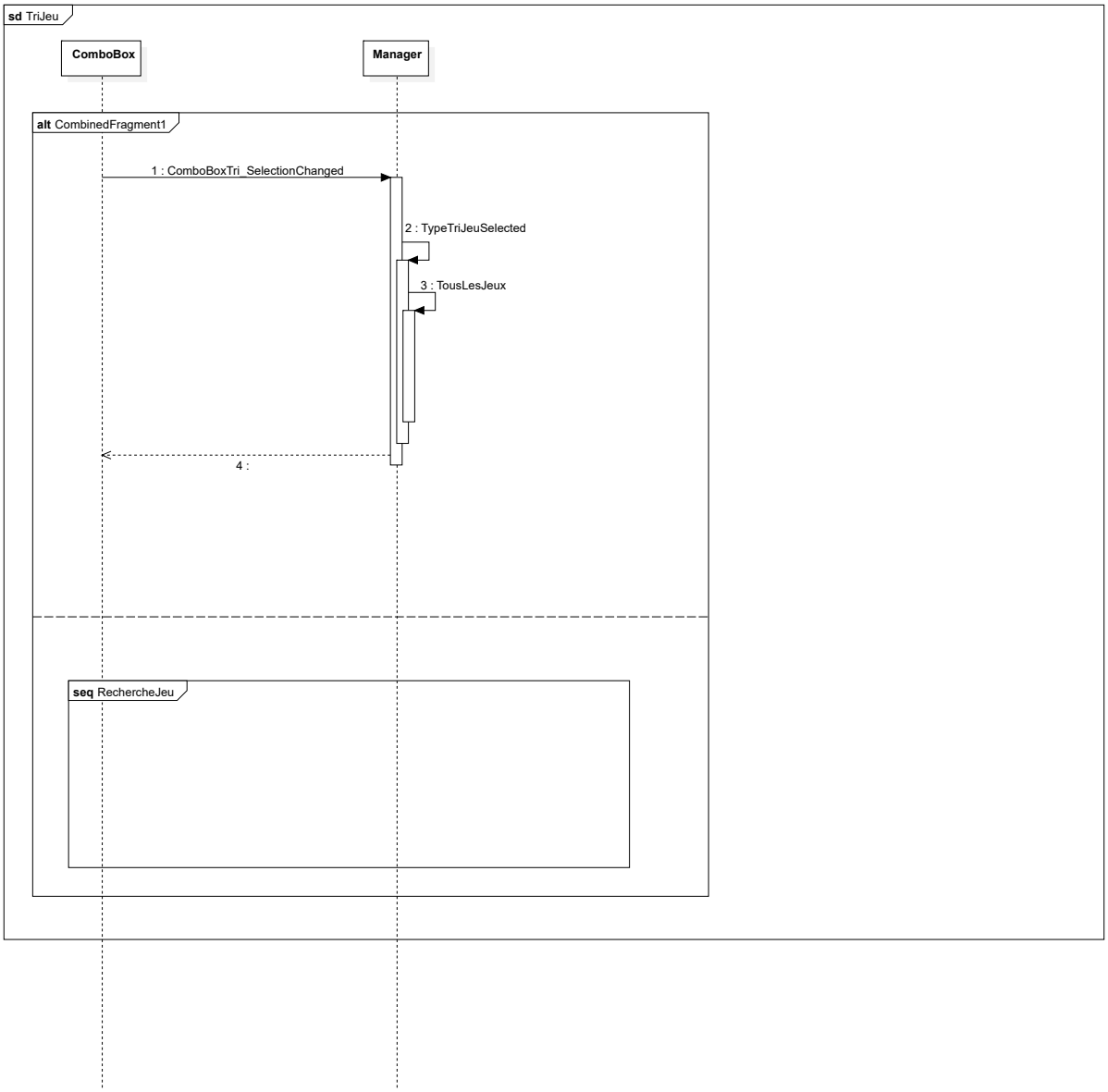


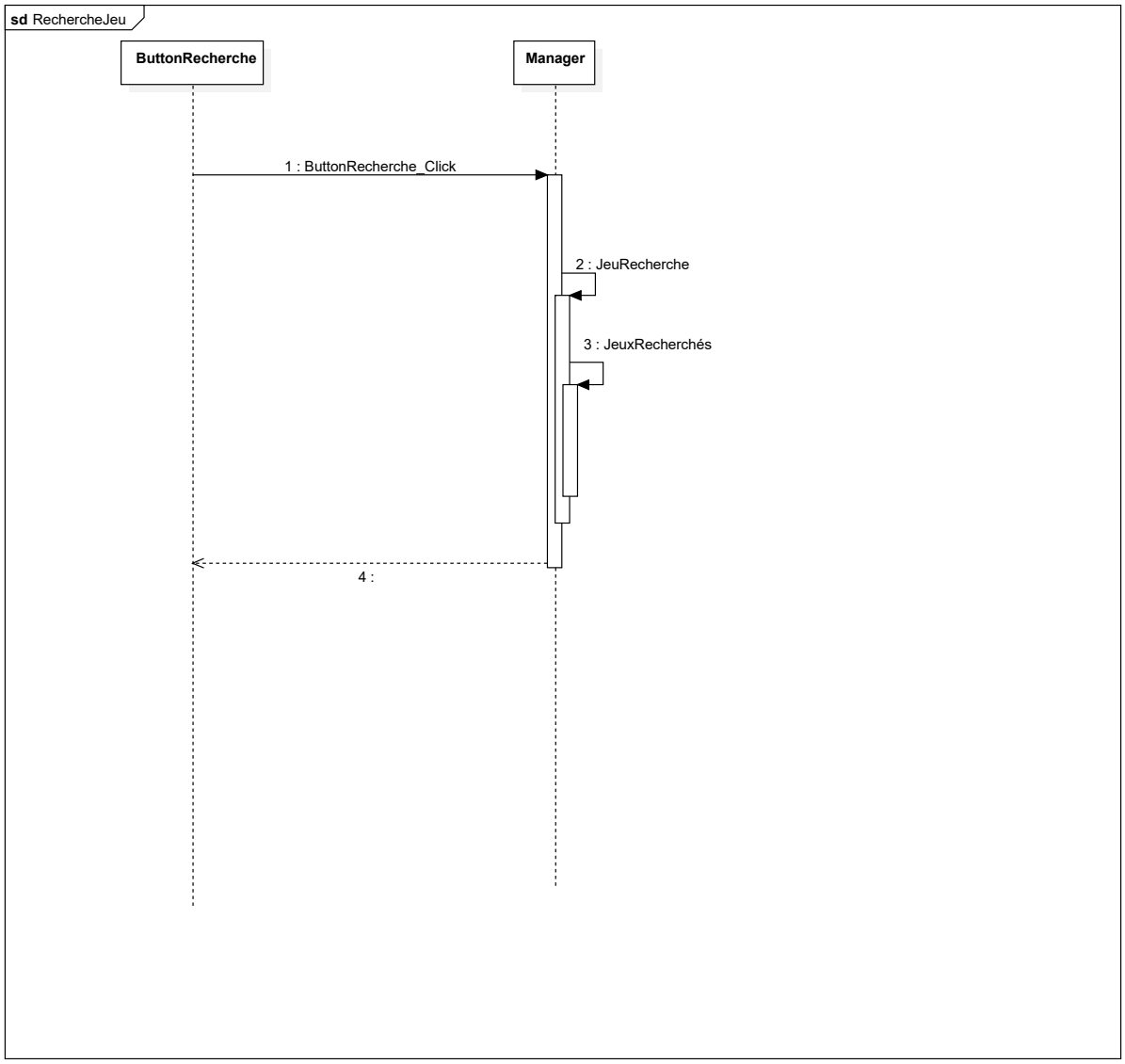


Description diagramme de séquence

Le diagramme de séquence montre le cas du tri. Dans le diagramme principal on peut voir la mise en situation de l'utilisateur. Quand il ne sélectionne rien dans la *comboBox* du tri, alors l'appelle de la Recherche se fait. En effet quand l'utilisateur ne choisit pas de type de tri, qui est ici *default,* alors ce n'est seulement une recherche qui peut être fait. La première séquence indique quand un type de tri n'est pas sélectionner quand le tri est fait. La deuxième indique le cas où l'utilisateur, via la *comboBox*, choisi un tri. Dans ce cas-ci le tri sera effectué. La séquence de ce diagramme fait référence au diagramme de séquence du *Tri*. Quand l'utilisateur clique sur la *combox* pour choisir les types tri. Cela engendre un événement. Cet événement va alors aller chercher dans le Manager le nom du type de tri dans la liste *TypeTriJeuSelected*. La propriété calculé *TousLesJeux* va alors trier les jeux en fonction du *TypeTriJeuSelected* sélectionné par l'utilisateur. Si ce n'est pas le cas, alors c'est la recherche qui s'effectue. Le troisième diagramme de séquence montre comment la recherche se fait. L'utilisateur écrit dans la *TextBox* et quand il appuie sur le *ButtonRecherche* cela engendre un événement. Celui-ci va chercher dans le Manager la propriété *JeuRecherche*. Cette propriété va permettre de faire la propriété calculée *TousLesJeux*. Dedans une liste *JeuxRecherchés* est créée. On ajoute chaque jeu correspondant à la recherche dans cette liste. Cette liste est ensuite retournée.


sd SequenceDiagram1
UserControlMain
Manager
seq TriJeu
[TypeTriJeu = default]
1 : comboBox
2 :
seq TriJeu
[TypeTriJeu != default]


sd TriJeu
ComboBox
Manager
alt CombinedFragment1
1 : ComboBoxTri_SelectionChanged
2 : TypeTriJeuSelected
3 : TousLesJeux
4 :
seq RechercheJeu


sd RechercheJeu
ButtonRecherche
Manager
1 : ButtonRecherche_Click
2 : JeuRecherche
3 : JeuxRecherchés
4 :