Manuscrit manager
Rapport de projet

Inès Carrasco Marie Elisabelar Florent Fayollas
Aurélie Forzani Mariem Jridi Nicolas Urien

Table des matières

Introduction

Se lancer dans la création d'une histoire est un défi que beaucoup tentent de relever, mais dont peu arrivent à bout. Bien souvent noyés par la multitude d'informations à retenir et à imbriquer entre elles, ces auteurs en herbe renoncent à la réalisation de leur projet.

Notre programme tente d'apporter une solution à ce problème spécifique. Le "Manuscrit Manager" est un outil ayant pour vocation d'aider le créateur d'histoires à organiser entre eux les éléments essentiels de sa narration. Grâce à un système de projets dans lesquels sont créés des modules, l'utilisateur dispose d'un espace organisé depuis lequel il a accès à de nombreux éléments cruciaux de son histoire. Il pourra, par exemple, enregistrer tous les détails relatifs à ses protagonistes et les relations qu'ils ont entre eux, noter puis visualiser les événements clés de son histoire, ou encore lister l'ensemble des lieux qui sont concernés par sa narration. Nous espérons ainsi faciliter le travail parfois complexe auquel se livrent les auteurs ou les scénaristes afin de leur laisser plus de temps pour se concentrer sur l'écriture de l'histoire.

1 Fonctionnalités principales de l'application

Le Manuscrit Manager offre de multiples possibilités à l'utilisateur pour lui permettre de bien gérer son projet. En effet, l'application en tant que telle est découpée en plusieurs fonctionnalités.

1.1 Gestion des projets

Dans cette partie, nous présentons les différentes fonctionnalités permettant la gestion du projet. Ces fonctionnalités ont été implantées pendant la première itération puisqu'elles sont cruciales dans la manipulation d'un projet.

1.1.1 Création d'un projet

En lançant l'application, l'utilisateur peut créer un nouveau projet vierge afin de commencer la rédaction de son projet tout en lui donnant un nom pour l'identifier.

Pour ajouter les modules qu'il souhaite, le menu `Créer un module` est fourni à l'utilisateur, lui présentant différents choix :

— `Événement`
— `Personnage`
— `Lieu`

1.1.2 Sauvegarde d'un projet

Cette fonctionnalité est primordiale pour l'utilisateur, afin qu'il puisse restaurer son projet et le rééditer à tout moment. L'utilisateur a l'opportunité de choisir le chemin de sauvegarde qu'il souhaite, avec la possibilité de le changer plus tard, grâce à l'action `Enregistrer sous`.

1.1.3 Chargement et affichage d'un projet

Après avoir enregistré son projet, l'utilisateur peut le charger plus tard en utilisant le menu `Fichier` puis en choisissant `Ouvrir` tout en précisant le chemin de sauvegarde sous lequel a été enregistré le projet. Il a, de plus, la possibilité d'adapter l'affichage des différentes fiches qu'il rajoute.

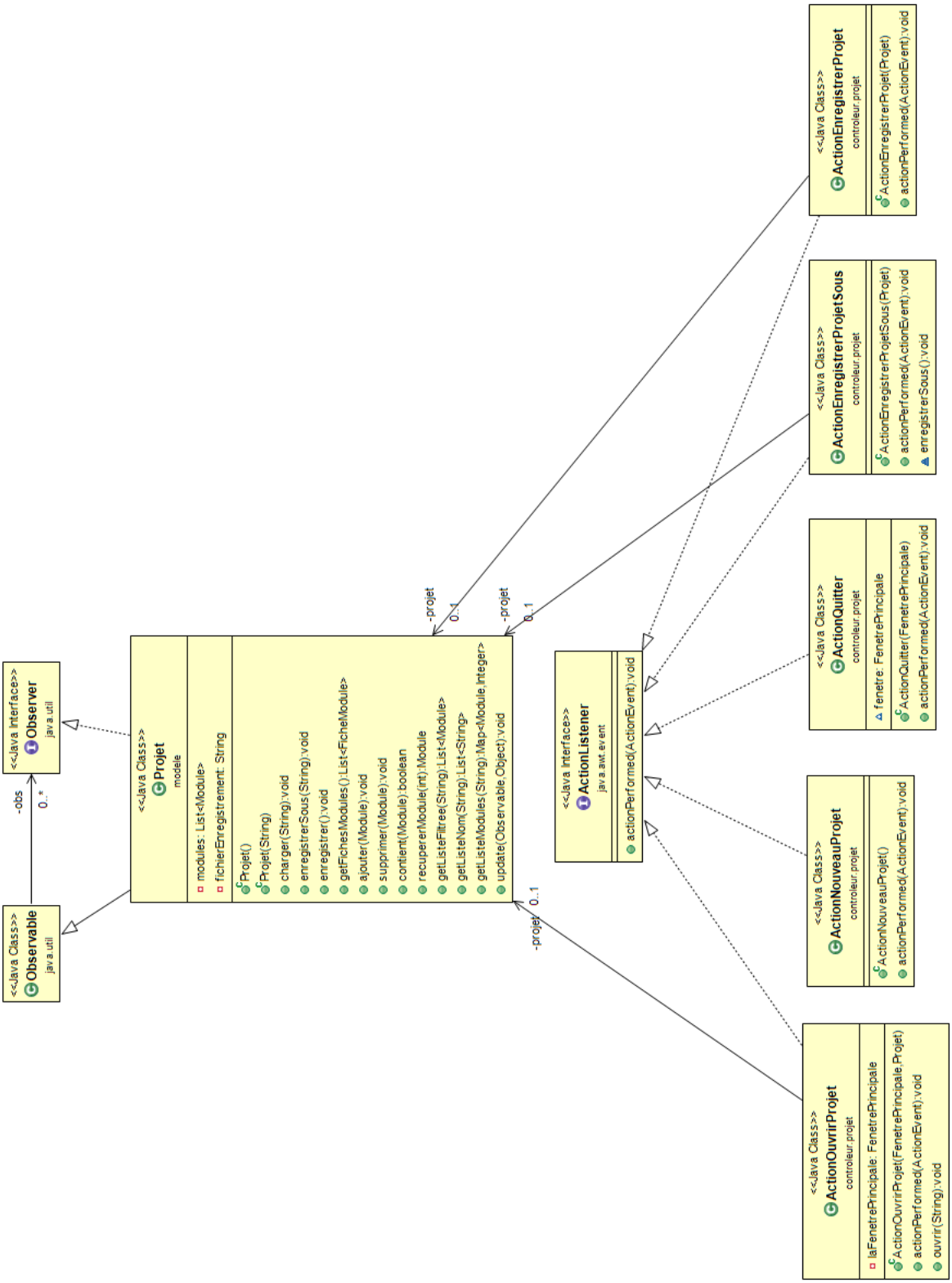

Figure 1 – Diagramme UML de la fonctionnalité Gestion d'un Projet

1.2 Gestion des personnages

Cette fonctionnalité permet à l'auteur de présenter toutes les informations liées à un personnage dans une fiche qui lui est associée, ainsi que les relations qu'il peut avoir avec les autres protagonistes. Deux boutons `Valider` et `Annuler`, qui ont été implémentés pendant la première itération, permettent respectivement de valider ou invalider les saisies en mémoire. Un bouton, implémenté à la deuxième itération, permet de supprimer le personnage.

1.2.1 Génération d'un personnage

Afin de générer un nouveau personnage, l'auteur peut créer une fiche personnage grâce au menu `Créer un module` et la sélection de l'option `Personnage`. Cette fiche lui présente différents champs à remplir pour définir son personnage à partir d'un nom, un prénom et, s'il le souhaite, une description associée pour présenter son caractère. C'était un point de départ pour traiter la gestion de personnages durant la première itération.

1.2.2 Spécification des relations entre personnages

Pendant la deuxième itération, nous avons commencé à améliorer la fiche personnage afin qu'elle offre à l'utilisateur un menu déroulant indiquant tous les personnages existants dans son roman. Souhaitant créer des liens avec les autres personnages, l'utilisateur pourrait alors choisir le personnage correspondant dans le menu déroulant, spécifier le type de relation dans le champ associé et avec un clic sur le bouton `Ajouter` la relation serait présente dans la zone dédiée. Cette tâche a été finie à la troisième itération.

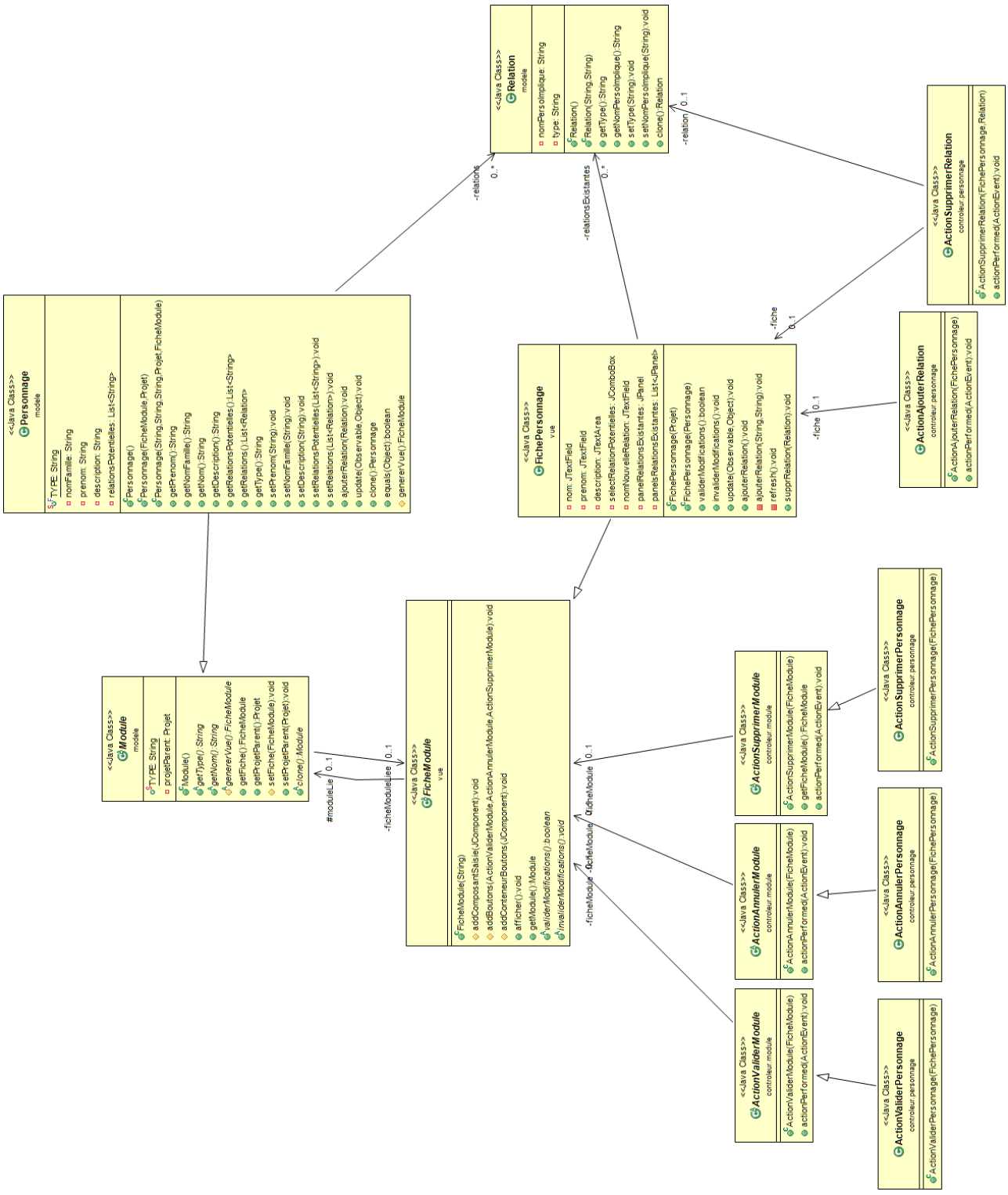

FIGURE 2 – Diagramme UML de la fonctionnalité Gestion des Personnages

1.3 Gestion de l'univers spatio-temporel

1.3.1 Gestion de l'univers spatial

Cette fonctionnalité, traitée dans la première itération, permet à l'auteur de définir un lieu à partir de la création d'une fiche associée, dans laquelle il remplit les différents champs présentés :

— Nom de lieu (obligatoire),

— Description des détails et caractéristiques de lieu (optionnelle).

De la même manière que la fiche `Personnage`, la fiche de lieu possède les différents boutons `Valider`, `Annuler` et `Supprimer`.

1.3.2 Gestion de l'univers temporel

Génération et gestion d'un événement Pendant la première itération, nous avons commencé à aborder cette fonctionnalité en générant une fiche d'événement. En effet, l'utilisateur a l'opportunité de créer un événement de son histoire en choisissant le menu `Créer un module` puis l'option `Événement`, ce qui lui permet de générer une fiche événement avec des champs vides. Il doit alors les remplir en fonction de ses choix. Les dates se remplissent avec un menu déroulant présentant les différentes possibilités de nombres de jours, de mois et d'année.

Comme les autres fiches, toutes les fiches d'événements disposent des différents boutons `Valider`, `Annuler` et `Supprimer`.

Lors de la deuxième itération, nous avons essayé d'améliorer cette fiche en rajoutant d'autres fonctionnalités. En effet, la fiche affiche aussi deux menus déroulants, l'un représente les personnages créés, l'autre les différents lieux existants afin de permettre à l'auteur de renseigner les personnages présents dans cet événement et les lieux qui lui correspondent. Cette tâche a été finie à la troisième itération.

Gestion de la frise chronologique Si l'utilisateur souhaite afficher l'ensemble de la trame chronologique et se référer rapidement et facilement dans son histoire, il peut générer une frise à partir des événements créés en cliquant sur le menu `Frise chronologique` et en choisissant l'option `Générer une frise à partir des événements`, s'il souhaite créer une frise vierge et la remplir ultérieurement il choisit l'option `Frise vierge`.

Cette fonctionnalité a été commencée pendant la deuxième itération et finie à la troisième itération. Cependant, par manque de temps lors de l'itération 3, il est impossible de rajouter des événements à une frise vierge.

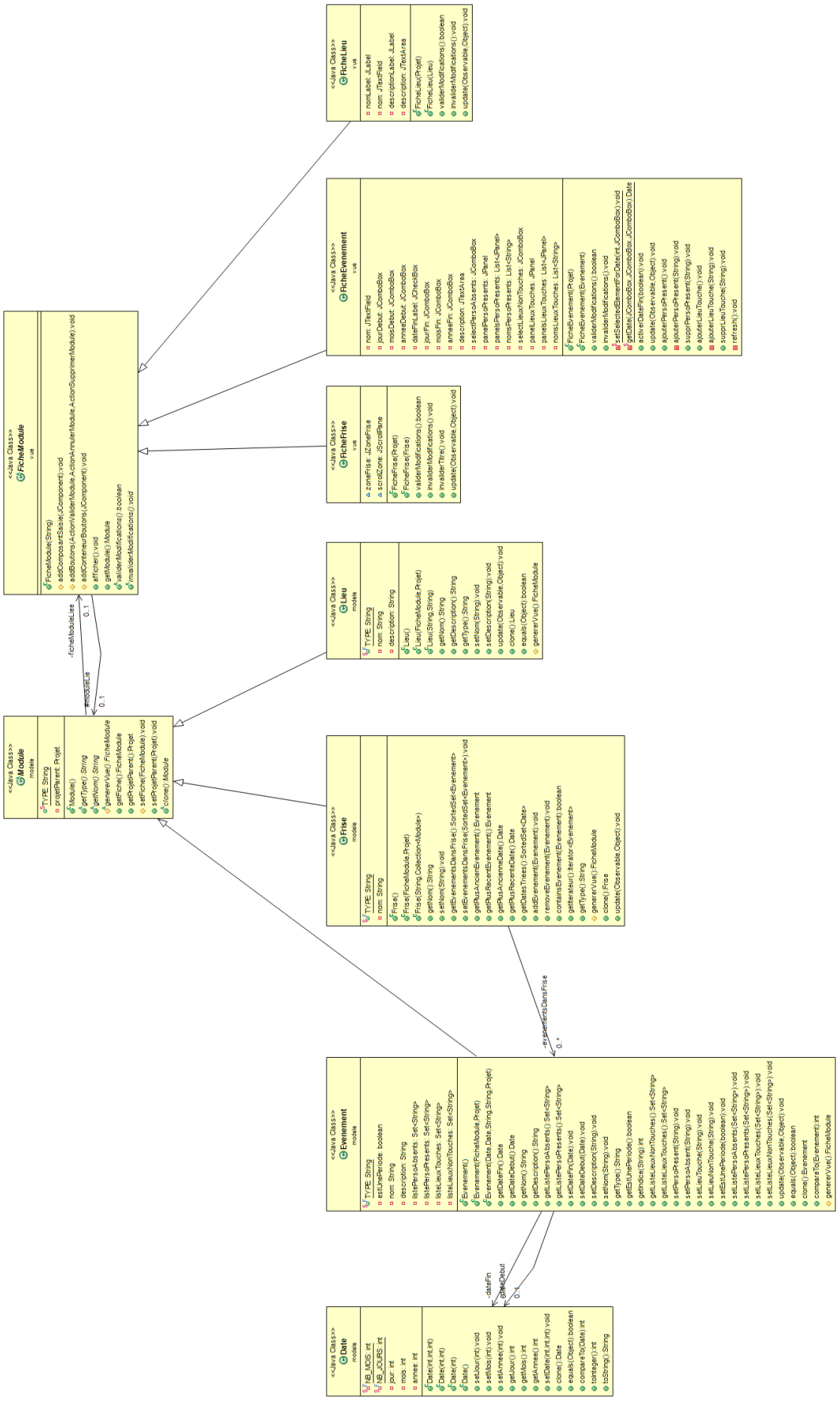

Figure 3 – Diagramme UML de la fonctionnalité Gestion de l'espace Spatio-Temporel (Vue et Modèle)

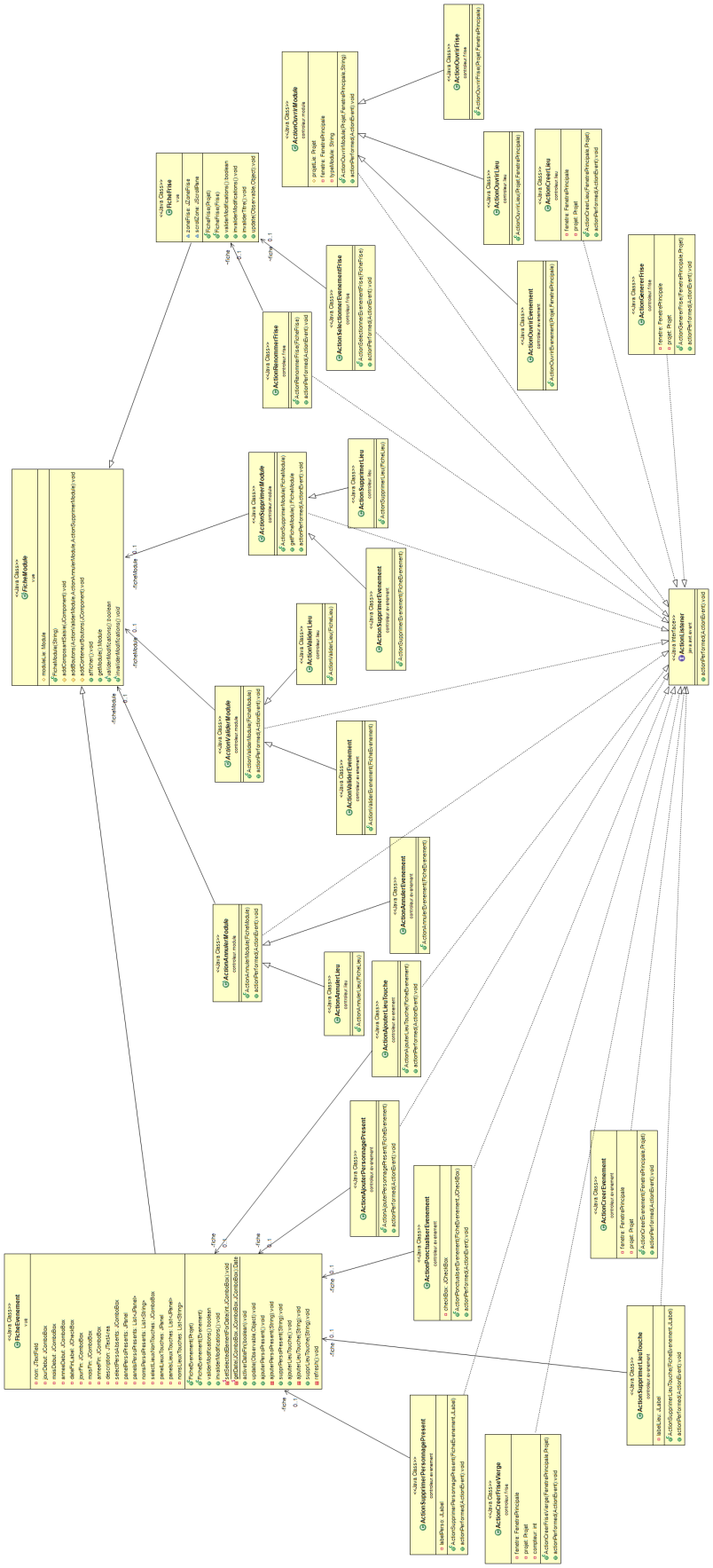

FIGURE 4 – Diagramme UML de la fonctionnalité Gestion de l'espace Spatio-Temporel (Contrôleur)

2 Présentation technique de l'application

Nous avons tout d'abord développé l'architecture basique du programme. En effet, nous avons développé dans un premier temps des vues simples, avec les modèles et contrôleurs associés, sans pouvoir tenir compte des liens pouvant exister entre les différents modules, par exemple entre un événement et un personnage. Nous avons ensuite implanté la gestion des liens entre les événements, lieux et personnages.

2.1 Architecture globale du programme

Le Manuscrit Manager est développé en utilisant le principe du patron Modèle-Vue-Contrôleur. Ainsi, le code de celui-ci est divisé en 3 paquetages principaux : le paquetage `modele`, le paquetage `vue` et le paquetage `controleur`.

2.1.1 Les modèles

Pour représenter les informations en mémoire, nous avons développé l'architecture de modèles décrite sur la figure 5, page 11. Ainsi, nous avons défini une classe abstraite `Module` qui permet de factoriser du code parmi tous les modules (personnage, événement, etc.). Elle est sérialisable –ce qui permet une sauvegarde, dans un fichier, de l'état actuel de l'objet en mémoire– et clonable. Elle est, de plus, observable et observatrice.

Tous les modules (`Personnage`, `Evenement`, `Lieu` et `Frise`) sont des classes héritant de `Module`. Ils définissent tous une constante permettant de connaître leur type sans faire appel à un `instanceof`.

Enfin, la classe `Projet` permet de gérer un projet. Elle permet donc les actions d'enregistrement (respectivement chargement) de données dans (respectivement depuis) un fichier, d'ajout d'un module au projet, de suppression d'un module au projet et de récupération de la liste des modules d'un certain type (par exemple, seulement les personnages).

2.1.2 Les vues

Toutes les vues graphiques des modèles sont définies dans les classes `Fiche*`. Encore une fois, la classe abstraite `FicheModule` permet de factoriser du code commun à toutes les fiches représentatives des modules.

Cette dernière hérite de `JInternalFrame`, ce qui permet d'afficher plusieurs fiches en même temps dans un `JDesktopPane` qui est géré par la fenêtre principale, décrite dans la classe `FenetrePrincipale`.

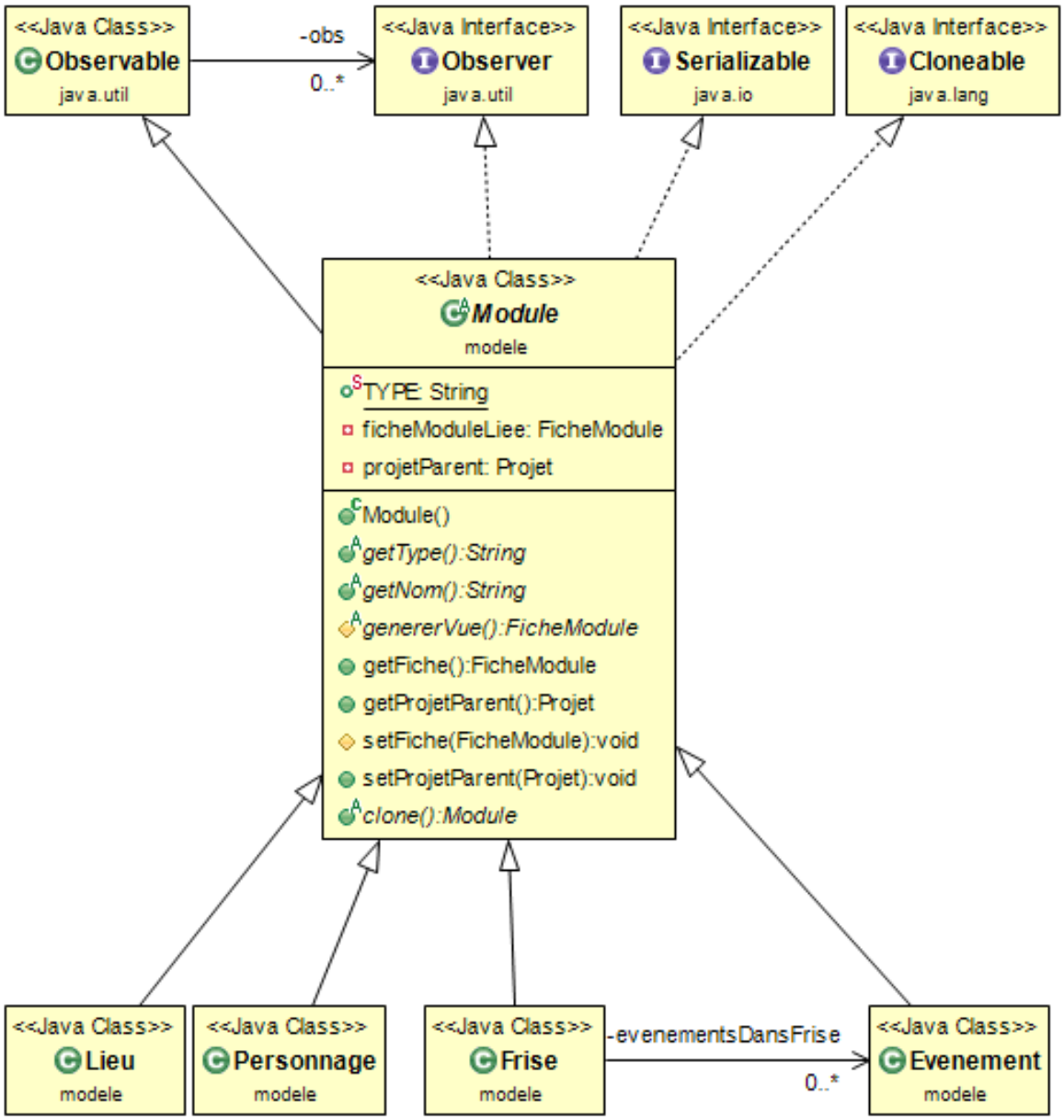

FIGURE 5 – Diagramme UML de l'architecture des modèles

2.1.3 Les contrôleurs

Les contrôleurs sont triés dans des sous-paquetages du paquetage `controleur`. Par exemple, les contrôleurs associés à la fiche personnage (`FichePersonnage`) se trouvent dans le paquetage `controleur.personnage`.

Une fois de plus, nous avons factorisé du code en implantant des contrôleurs abstraits contenus dans le paquetage `controleur.module`.

Création d'un module La création d'un module est faite par création d'une fiche correspondante au module demandé. Cette fiche est ensuite ajoutée à la fenêtre principale. L'utilisateur peut alors remplir les informations qu'il juge nécessaire, mais doit absolument remplir les champs "Nom" et/ou "Prénom".

Validation des modifications apportées Lorsque l'utilisateur clique sur le bouton `Valider` de la fiche module en cours d'édition, l'algorithme suivant est appliqué :

1. Appeler la méthode `validerModifications` de la fiche module. Cette méthode permet de tenter d'enregistrer les saisies de la fiche dans le modèle du module associé à la fiche. Elle renvoie `true` si la modification a bien été faite.
2. Si celle-ci renvoie `true` :
   (a) Ajouter le module (associé à la fiche) au projet, si ce dernier ne le contient pas déjà,
   (b) Appeler la méthode `update` du projet pour prévenir l'ensemble des modules qu'une modification a été effectuée,
   (c) Appeler la méthode `invaliderModifications` de la fiche module, qui permet de recharger les informations à afficher dans la fiche, depuis le modèle du module associé à la fiche.

   Sinon : ne rien faire

Annulation des modifications apportées Lorsque l'utilisateur clique sur le bouton `Annuler`, le contrôleur appelle la fonction `invaliderModifications` de la fiche qui lui est associée. Cette dernière charge alors les informations qu'elle doit afficher depuis le modèle du module qui lui est associé.

Dans le cas où l'utilisateur clique sur le bouton `Annuler` alors qu'il s'agit de la création d'un module, le contrôleur appelle la méthode `dispose` de la fiche module, afin de la fermer complètement.

Fermeture d'une fiche module Lorsque l'utilisateur ferme une fiche module en cliquant sur la croix en haut à droite de la fiche, la fiche n'est que masquée.

Ouverture d'une fiche module fermée Lorsque l'utilisateur choisit d'ouvrir une fiche module précédemment fermée, la fiche est simplement mise visible par un appel à `setVisible(true)`.

2.2 Gestion des liens entre modules de types différents

Comme indiqué précédemment, les modules sont observables et observateurs. Ces deux propriétés sont nécessaires pour prendre en compte les modifications apportées à un module particulier et les répercuter sur les fiches modules et dans les modèles.

Par exemple, un événement doit répertorier des lieux concernés par cet événement. Ainsi, lorsqu'un lieu est ajouté, supprimé ou modifié, tous les événements doivent le savoir et prendre en compte cette action.

Pour pouvoir réaliser cela, nous avons implanté la solution suivante :

— Un projet observe les modules lui appartenant,
— Un module observe son projet parent,
— Une fiche module observe le module qui lui est associé,
— Lors de la transmission de l'information de modification (avec `notifyObservers`), un objet du type `Notification` est transmis.

La classe `Notification` permet de transmettre une information sur l'ajout, la modification ou la suppression d'un module. Ainsi, elle contient 3 attributs :

1. `ancienEtat` qui contient l'ancien état de l'objet modifié ou supprimé,
2. `nouvelEtat` qui contient le nouvel état de l'objet modifié ou ajouté,
3. `type` qui contient une information permettant de déterminer si la notification concerne un ajout, une modification ou une suppression de module.

Cependant, avec la solution précédente, il y avait une boucle d'observation. En effet, lorsqu'un module est modifié, le projet en est informé. Il informe alors tous ses modules fils. Ceux-ci prennent en compte la notification et informent qu'ils ont été changés pour prévenir leur fiche associée du changement. Alors, le projet est aussi informé et la boucle est créée.

Nous avons donc gardé le modèle d'observation précédent, mais en appliquant la solution suivante :

— Lorsqu'un module est ajouté, modifié ou supprimé, on appelle la méthode `update` du projet,
— Lorsqu'un module reçoit une notification du projet, il la transmet à sa fiche associée en appelant la méthode `update` de la fiche module.

Avec le recul, nous avons compris que nous aurions pu implanter une solution plus simple :

— Le projet n'observe personne,
— Un module observe son projet parent,
— Une fiche module observe le projet parent,
— Lors de la transmission de l'information de modification (avec `notifyObservers`), un objet du type `Notification` est transmis,
— Lorsqu'une modification de module est faite, le projet en est informé par l'appel d'une méthode publique.

2.3 Chargement et sauvegarde d'un projet

Les méthodes de sérialisation Java permettent d'enregistrer des données depuis un fichier sur le disque dur. Les méthodes de désérialisation permettent de faire l'action inverse.

Pour enregistrer et charger un projet dans/depuis un fichier, nous utilisons les classes `java.beans.XMLEncoder` et `java.beans.XMLDecoder`. Celles-ci permettent de sauvegarder et charger des données depuis un fichier XML.

Nous aurions pu simplement utiliser les méthodes de sérialisation basique. Cependant, celles-ci génèrent et chargent un fichier de sauvegarde binaire. Ainsi, il aurait été complexe de vérifier le bon enregistrement des données. Pour pouvoir utiliser ces méthodes de sérialisation, les classes devant être sérialisées doivent avoir un constructeur par défaut et des getters et setters pour chacun des attributs à (dé)sérialiser.

2.4 Particularités implantées

2.4.1 Classe `modele.Date`

Lors du développement du modèle `Evenement`, nous avons rencontré un problème majeur. En effet, d'après le cahier des charges et les User Stories définies, l'utilisateur devait avoir la capacité de saisir une date en donnant :

— soit une année,
— soit une année et un mois,
— soit une année, un mois et un jour.

Cependant, la classe `java.util.Date` ne permettait pas cela. Nous avons donc implanté une classe `modele.Date` qui permet ce cas d'utilisation. Cependant, celle-ci ne prend pas en compte le calendrier réel. Ainsi, il est possible pour l'utilisateur de saisir une date telle que le "31 Février 2016", alors que cette date (le 31 février) n'existe pas.

2.4.2 Classe `assets.DesktopScrollPane`

Nous utilisons cette classe dans le but de gérer le `JDesktopPane` de manière plus conviviale pour l'utilisateur. En effet, lorsque nous n'utilisions pas cette classe, si l'utilisateur réduisait la fenêtre principale, celui-ci pouvait ne plus voir certaines des fiches modules qu'il avait créé. Grâce à cette classe, lorsque l'utilisateur réduit la taille de la fenêtre, des barres de défilement apparaissent.

Nous avons trouvé cette classe sur le site :
`https://github.com/dukke/swing-desktopScrollPane`

2.4.3 Classe `assets.SpringUtilities`

Nous utilisons cette classe pour gérer les layouts de type `SpringLayout` dans les fiches modules. Nous l'avons récupérée depuis le site officiel de Java :
`http://docs.oracle.com/javase/tutorial/uiswing/examples/layout/SpringGridProject/src/layout/SpringUtilities.java`

2.4.4 Classe `Frise`

Le modèle `Frise` contient simplement une collection ordonnée des événements à tracer dans la frise ainsi qu'un nom permettant d'identifier la frise.

La vue `FicheFrise` permet la création d'une fiche module simple contenant une zone centrale du type `JZoneFrise` qui est un `JPanel` dans lequel est tracé la frise chronologique.

Les constantes définies au début du fichier `JZoneFrise` permettent de régler la génération graphique de la frise chronologique.

3 Organisation de l'équipe Agile

3.1 Décomposition du Manuscrit Manager

Au départ, nous avons décomposé l'application en 6 fonctionnalités, comportant chacune entre 3 et 6 Epics. Puis nous avons classé ces Epics selon leur importance et leur plus-value pour l'utilisateur. Nous avons adopté le classement suivant :

— le niveau 1 correspond à une Epic qui doit être traitée en priorité,
— le niveau 2 correspond à une Epic qui doit être traitée en un second temps,
— l'absence de niveau indique que l'Epic n'est pas à considérer pour le moment.

Toutes les fonctionnalités prévues initialement n'ont pas pu être traitées. Par exemple, la fonctionnalité concernant la gestion spatiale et temporelle a largement été privilégiée aux dépens de la fonctionnalité sur l'objectif d'écriture. En effet, elle représente à nos yeux un plus grand intérêt pour l'utilisateur de l'application.

Puis, nous avons décomposé ces Epics en User Stories. C'est sur cette décomposition que nous nous sommes appuyés pour répartir le travail entre les différentes itérations, en respectant, bien entendu, les priorités fixées au niveau des Epics. A la fin de la première itération, nous nous sommes fixé une échelle de points difficulté et de points valeur métier, afin d'estimer la charge que notre équipe agile est capable de produire pendant une itération. Pour cela, nous avons scindé l'équipe en deux groupes. Chacun des groupes devaient attribuer les points de difficulté pour l'une, et de valeur métier pour l'autre, à chacune des User Story. Ainsi, nous avons pu estimer ce que l'équipe pouvait produire durant les deuxième et troisième itérations.

3.2 Fonctionnement du groupe agile

Avant la première itération, nous avons procédé à une longue réunion pour déterminer l'architecture globale de l'application. Cette réunion s'est conclue par une répartition des User Stories.

Au cours de chaque itération, nous organisions une grosse réunion d'environ 2 heures pour échanger sur l'avancée de chacun des membres, sur ce qui avait été fini, sur ce qu'il était envisageable de faire. Lors de ces réunions, nous échangions également sur les idées de conception, sur comment chacun imaginait sa partie, sur les points de blocage et sur les solutions apportées. Puis, au cours de l'itération, nous organisions une autre réunion, bien plus courte (environ 30 minutes) au cours de laquelle chacun expliquait ce qu'il avait fait, et ce qu'il envisageait.

Suivant la disponibilité ou la difficulté de l'itération, une autre courte réunion était organisée en fin de semaine.

Conclusion

L'application finale que nous proposons permet de construire un univers complet. Elle permet de se repérer dans le temps, dans l'espace et de définir les différents protagonistes d'une histoire.

Cependant, l'application n'est pas aussi complète que nous l'avions imaginée au départ. Elle devait à l'origine permettre de réaliser la saisie de texte, de gérer le temps de travail et de générer des cartes mentales.

Ces améliorations restent cependant possibles. Elles correspondent aux fonctionnalités ayant soit des valeurs ajoutées moindres soit des niveaux de difficultés important.

Nous avons donc été bien trop ambitieux au départ.

Cela nous a permis d'expérimenter par nous-mêmes les principes exposés lors de nos cours concernant les méthodes agiles. Il est en effet difficile d'estimer le temps nécessaire à l'élaboration d'une application parfaite cependant, en se fixant un temps de travail précis, on peut aboutir à une application contenant les principales fonctionnalités voulues.

Malgré les fonctionnalités éliminées, le "Manuscrit Manager" remplit finalement l'objectif principal que nous nous étions fixé.