HomePlans - Rapport
INFO-F-307

Titouan Christophe
Pierre Gérard
Florentin Hennecker
Walter Moulart
Bruno Rocha Pereira
Julian Schembri

17 février 2015

Table des matières

Chapitre 1

Itération 1

1.1 Introduction

Nous avons décidé de développer l'histoire 1 proposée par le client lors de cette phase. Ce choix a été fait dans le but de fournir le plus vite possible un socle pour les fonctionnalités à suivre, et se présentait logiquement comme le seul choix possible.

Vous pourrez lire dans la suite de ce chapitre la description des choix qui ont été faits quant aux librairies que l'équipe a décidé d'utiliser, suivie d'un rapport sur cette phase.

1.2 Librairies introduites

1.2.1 Enregistrement du projet

Utilisation de SQLite pour enregistrer les projets, à l'aide de ORMlite, et ses dépendances (ormlite-jdbc, ormlite-core, sqlite-jdbc)

- sqlite a les avantages d'une DB relationnelle, mais enregistre dans des fichiers : facilité de déplacement des projets, pas besoin de serveur
- sqlite permet d'avoir une DB en mémoire vive : pratique pour les tests
- sqlite est utilisable dans de nombreux autres langages : augmente la productivité en permettant d'éditer les fichiers de projets dans le langage préféré de chaque dev
- ORM facilite l'accès aux données (Design pattern DataMapper http ://martinfowler.com/eaaCatalog/dataMapper.html)
- ORMLite permet d'utiliser plusieurs types de bases de données différentes à travers JDBC. Possibilité donc d'utiliser autre chose qu'SQLite si nécessaire.

1.2.2 Librairie GUI

Les requirements de l'interface graphique sont les suivants

- Afficher une fenêtre esthétique
- Mettre des menus / boutons
- Incorporer une vue 2D/3D
- Compatible avec la lib 3D choisie

Les différentes possibilités de GUI sont :

- AWT (Abstract Window Toolkit)

– Swing
– SWT (Standard Widget Toolkit)
– JavaFX
– Apache Pivot
– Qt Jambi

Swing

Nous avons retenu **Swing**.
– Customisable : Oui
– Léger : Non
– Cross-Platform : Oui
– Documentation : Complet
– Faiblement couplé
– Suit le design pattern MVC
– Compatible avec jMonkeyEngine

Toutes ces raisons font qu'on utilisera Swing comme librairie pour construire notre interface graphique. Voici un aperçu des autres libraires éligibles, et les raisons pour lesquelles elles ont été refusées.

AWT

– Customisable : Non
– Léger : Oui
– Cross-Platform : Oui
– Documentation : Complet
– AWT utilise les objets du système
– Swing hérite des objets de AWT

On n'utilisera pas AWT car il n'est pas assez customisable.

SWT

– Customisable : Non
– Léger : Oui
– Cross-Platform : Oui
– Documentation : Très complet
– Développé par l'équipe d'Eclipse
– Se place un peu comme premier concurrent de Swing

On n'utilisera pas SWT car il n'est pas assez customisable.

Java FX

On n'utilisera pas Java FX car ils se sont spécialisés dans les interfaces d'application web et ce n'est pas forcément utile ici.

Apache Pivot

On n'utilisera pas Apache Pivot pour les mêmes raisons que Java FX.

Qt Jambi

- Customisable : Oui
- Léger : Non
- Cross-Platform : Oui mais limité à QT4.6
- Documentation : Mauvaise
- Intégré à Eclipse
- Assez complet
- Accès a une database sql incluse

Inconvénients :

- Pas de doc sur le site QtJambi
- Compliqué pour l'intégration de la 3D . Il existe Qt3D mais on s'est dirigé vers Jmonkey
- Bloqué a la version 4.6

1.2.3 Librairie 3D

Nous choisissons d'utiliser jMonkeyEngine comme librairie 3D pour ces raisons :

- Elle est sous license BSD
- Une des interfaces les plus high level
- Community driven
- Développement encore actif
- Documentation extensive
- Engine complet
- Refonte complête de JME2 pour le mieux
- Intégration facile dans un GUI swing : http ://hub.jmonkeyengine.org/wiki/doku.php/jme3 :advanced :swing_canvas

Justification de l'écartement d'autres librairies :

JOGL :

- Utilise SWT comme système de fenêtrage

GL4Java :

- Vieux et obsolète

Java3D :

- Abandonné

Ardor3D :

- Prise en main difficile et risque de requérir beaucoup plus de temps

1.3 Histoires utilisateur

1.3.1 Enregistrement d'un projet

Durée estimée : 20h | **Difficulté estimée : 2/3** | **Assigné à : Titouan** | Groupe **3**

Les projets d'architecture doivent être enregistrables dans des fichiers, et il doit être possible de charger un projet depuis un fichier. Différents éléments doivent être enregistrés, comme la configuration du projet (nom, auteur, ...), ainsi que des éléments affichables (polyèdres pour le moment, éventuellement les tags par la suite).

Les projets doivent être enregistrés séparément et sélectionnables dans un navigateur de fichiers pour ouverture ou enregistrement.

Statut : fait | **Remarques :**

1.3.2 Interface graphique

Durée estimée : 30h | **Difficulté estimée : 1/3** | **Assigné à : Julian et Pierre** | Groupe **3**

L'interface graphique sera composé d'un menu, d'une barre d'outils et d'un écran splitter en deux. Sur cet écran splitter il y aura à gauche les objet utilisés et a droite le rendu 2D et 3D.

Statut : fait | **Remarques :**

1.3.3 Affichage du monde 2D et 3D

Durée estimée : 10h | **Difficulté estimée : 1/3** | **Assigné à : Florentin, Bruno et Walter** | Groupe **3**

L'utilisateur devra pouvoir choisir d'afficher le plan d'architecture autant en 3D qu'en 2D. Chaque pièce pourra avoir une forme différente et n'aura pas forcément la forme d'un rectangle.

Statut : fait | **Remarques :** Cette tâche a pris plus de temps que prévu à cause de notre perfectionnisme pour les caméras différentes, le look and feel de la vue et quelques détails qui ont amélioré l'usabilité de l'application

1.3.4 Navigation dans le monde

Durée estimée : 5h | **Difficulté estimée : 1/3** | **Assigné à : Bruno et Julian** | Groupe **3**

Dans la vue 3D, l'utilisateur devra avoir le contrôle de la caméra, que ce soit pour zoomer ou déplacer la caméra sur les cotés, et ce, aussi bien grâce à la souris que grâce aux flèches.

Statut : fait | **Remarques :**

1.3.5 Modification de la géométrie d'une pièce

Durée estimée : 40h | **Difficulté estimée : 3/3** | **Assigné à : Walter, Titouan, Bruno, Florentin** | Groupe **3**

L'utilisateur a accès à une vue d'édition lui permettant de rajouter ou éditer des murs et des étages.

Tous les points du "sol" sont à la même hauteur, et à chacun d'eux est associée une élévation.

La hauteur de l'étage est déterminée par son point le plus haut. Le sol de l'étage suivant est égal à la hauteur du point le plus haut de l'étage précédent

Statut : fait | **Remarques :**

1.3.6 Création d'un projet de demo

Durée estimée : 5h | **Difficulté estimée : 1/3** | **Assigné à : Walter et Julian** | Groupe **3**

Lorsque l'utilisateur lance pour la première fois le programme EA3D, un projet de démonstration complet s'ouvre, pour lui montrer tout ce qu'il est possible de faire.

Statut : fait | **Remarques :**

1.3.7 Enregistrement automatique d'un projet

Durée estimée : 2h | **Difficulté estimée : 1/3** | **Assigné à : Titouan et Pierre** | Groupe **3**

Le projet sur lequel travaille l'utilisateur est sauvé automatiquement sur le disque. L'utilisateur peut aussi enregistrer une copie du projet, auquel cas un nouveau fichier de projet est créé, et devient le projet courant.

Par défaut, lorsque l'utilisateur ouvre le programme, le dernier projet utilisé lui est présenté.

Statut : fait | **Remarques :** Il faudra peut-être revoir le système car les performance en écritures/lectures fréquentes du disque ne sont pas excellentes.

1.3.8 Intégration de la vue 3D dans la GUI

Durée estimée : 10h | **Difficulté estimée : 2/3** | **Assigné à : Pierre** | Groupe **3**

Il faut intégrer jmonkey qui s'occupe de la 3d dans swing qui s'occupe de l'interface graphique

http ://hub.jmonkeyengine.org/wiki/doku.php/jme3 :advanced :swing_canvas

Statut : fait | **Remarques :**

1.4 Rapport de fin d'itération

Le développement de l'itération 1 s'est bien déroulé. La charge de travail a été correctement répartie le long des 3 semaines de développement, tout le monde a pris part de manière équilibrée au travail et on ne déplore aucun conflit.

1.4.1 Suivi du planning

Toutes les sous-histoires de l'itération 1 ont été implémentées et nous estimons avoir réussi à le faire tout en gardant une architecture propre.

Le temps étant court et la fonctionnalité demandée étant plutôt grande, nous avons d'abord commencé en petits groupes pour la découverte des outils et librairires que nous allions utiliser. Nous avons ensuite essayé de tout mettre ensemble pour tester le produit complet à un stade pré-alpha.

C'est à ce moment que nous avons effectué un premier refactoring pour pouvoir tous continuer sur les différents aspects du projet dans une architecture propre.

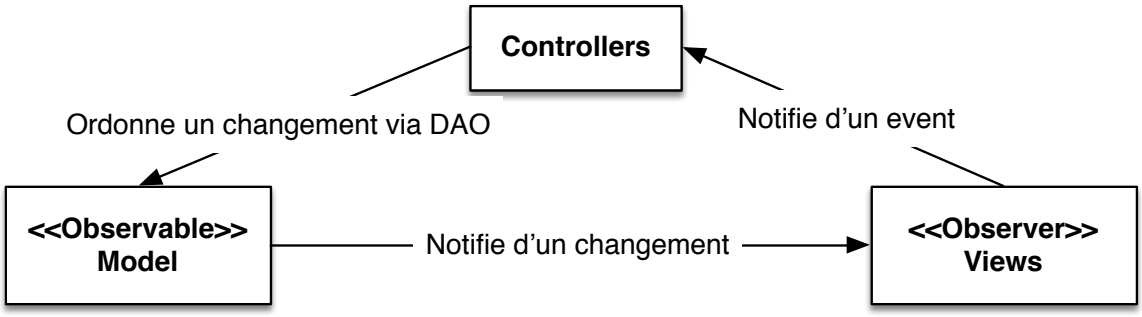

FIGURE 1.1 – Diagramme de composants général

Timesheet

Tâche	Bruno	Florentin	Julian	Pierre	Titou	Walter
Management	6,00	19,00	6,00	6,00	6,11	6,00
Découverte lib 3D	7,00	5,75				
Compilation projet Eclipse/Ant			1,00	1,00	5,75	
Découverte GUI (swing)			5,00	8,33		1,00
Model / Découverte ORMlite					4,00	
Refactoring/Debug		8,00		11	3,00	
1.1 Enregistrement d'un projet					18,00	
1.2 GUI	6,33		4,00	25,83	5,00	4,00
1.3 Affichage du monde 2D/3D	15,00	11,00	9,00		2,00	12,00
1.4 Navigation dans le monde 3D	12,00					2,00
1.5 Modif. de la géométrie des pièces		8,00			6,00	12,00
1.6 Création d'un demo project						
1.7 Enregistrement automatique				3	2,00	
1.8 Intégration GUI+3D	4,00	2,00	3,00	1,00	5,00	

1.4.2 Architecture

Architecture générale

On voit directement l'utilisation du MVC, couplé avec un design pattern Observable et un DAO. Cette architecture nous permet de :

- rajouter/supprimer une vue (+ contrôleur) très facilement
- garder toutes les vues à jour
- découpler un maximum toutes les parties de l'application
- garder l'état du projet entre deux sessions d'utilisation

Prenons un exemple : le passage d'une vue 3D à une vue 2D et inversement. Ce passage a des implications dans plusieurs vues. La première est bien évidemment l'éditeur général, qui doit afficher le monde différemment en fonction du mode ; la deuxième est la toolsbar, qui permet de changer de mode.

Quand on clique sur le bouton 2D/3D, la ToolsBarView notifie son contrôleur d'un event reçu. Le contrôleur utilise alors le DAO pour enregistrer que le mode a été changé. Le modèle a en effet une valeur de configuration qui stocke le mode. Ce dernier notifie

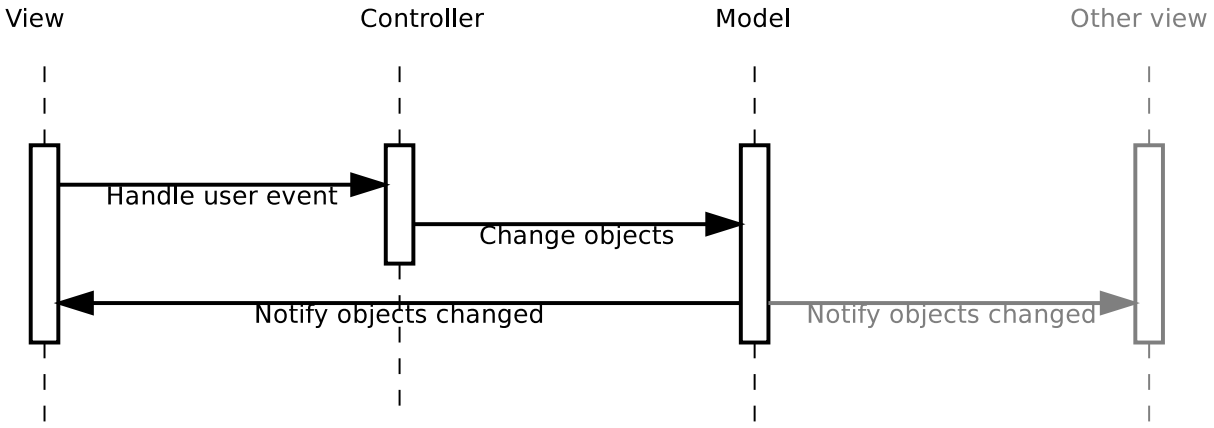

FIGURE 1.2 – Diagramme de séquence MVC

alors toutes les vues concernées du changement (via le pattern Observer), dont l'éditeur principal qui va alors changer son mode d'affichage.

Architecture du modèle

Le modèle est architecturé selon le design pattern Data Access Object. La classe `Project` a la responsabilité de l'accès au fichier du projet, et des valeurs de configurations générales. Il donne en outre accès au `GeometryDAO`, qui est un singleton pour le projet, et qui permet d'effectuer les opérations CRUD [1], ainsi que des recherches d'objets selon différents critères.

Les objets enregistrables par le `GeometryDAO` implémentent tous l'interface `Geometric`. On distingue 4 catégories d'objet dans le modèle (Figure 1.3) :

- Les `Point` représentent une position dans l'espace
- Les `Shape` définissent des formes bidimensionnelles composées de `Point`
- Les `Grouped` définissent des constructions sur les `Shape`
- Les `Floor` permettent de grouper ces éléments par étage.

La séparation de la forme et des constructions nous permet d'associer facilement différents éléments de la pièce (murs, sols, plafonds) qui sont bâtis à partir des mêmes points. En outre, plusieurs pièces peuvent contenir le même point, facilitant ainsi la détection de pièces adjacentes, le déplacement d'un coin de mur entre plusieurs pièces, ...

Architecture de la GUI

L'interface graphique est composée de plusieurs vues qui ont chacune leur contrôleur propre. La classe GUI est celle qui gère la fenêtre et les composantes de premier niveau :

- la barre de menus
- les popup de sélection de fichiers
- la barre d'outils
- l'éditeur principal (qui gère la vue 3D/2D et la TreeView)

1. Create Refresh Update Delete

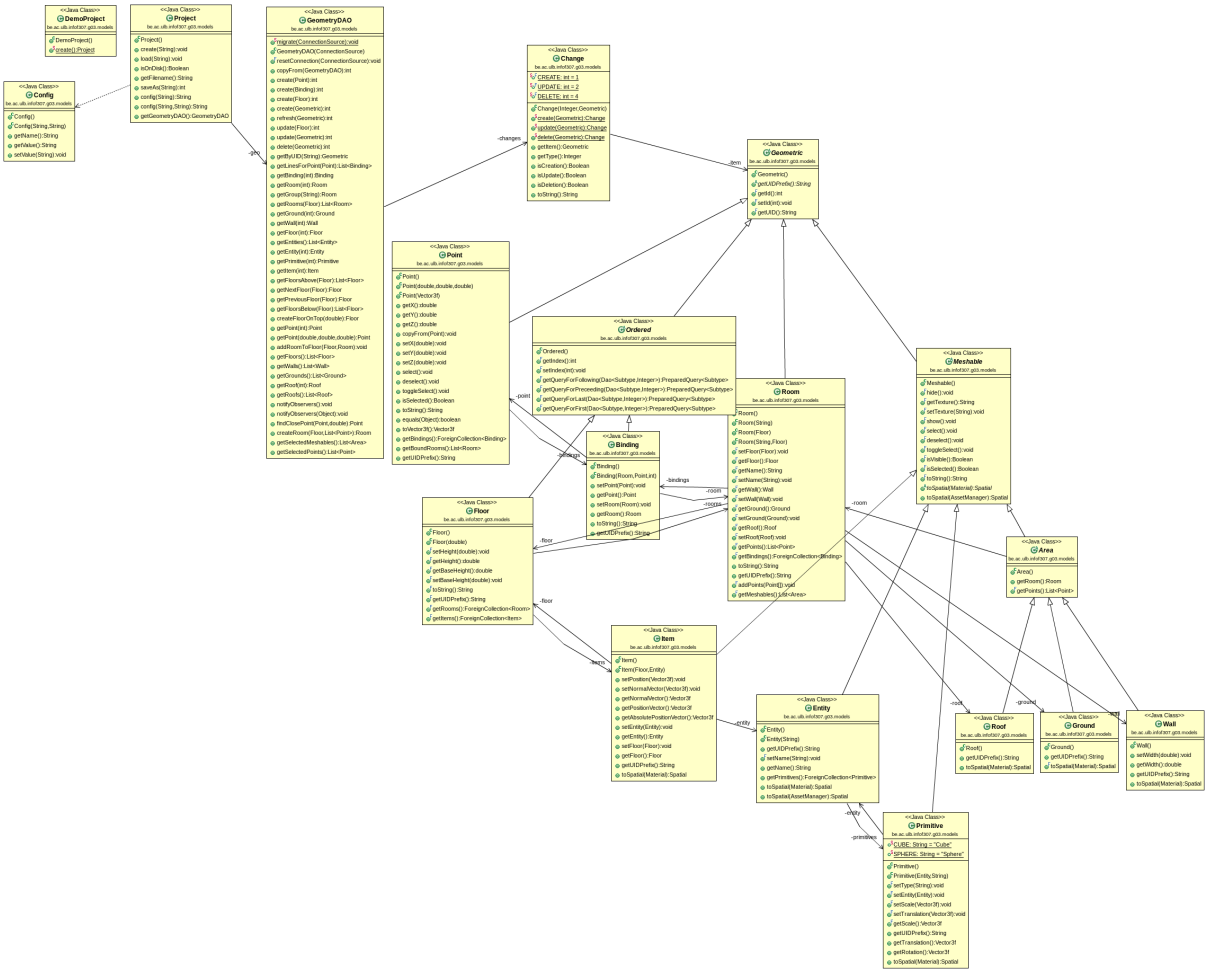

FIGURE 1.3 – Diagramme de classes du modèle

La WorldView est une sous-classe d'une SimpleApplication jMonkey. C'est le canevas 3D à proprement parler.

1.4.3 Bonnes pratiques utilisées

Plusieurs pratiques ont été mises en place pour faciliter le développement en groupe, comme par exemple le pair programming ou les sprints d'une journée.

Pair programming

À plusieurs moments, nous avons travaillé par deux sur un même ordinateur pour les problèmes plus épineux. Un développeur écrivait du code, et l'autre essayait de le corriger et de penser aux edge cases en même temps. Cela nous a permis de partager beaucoup plus facilement des idées et de compléter des features complexes de manière rapide et robuste.

Sprints d'une journée

En déployant un outil de statistiques de notre repository Git, on voit que le samedi est le jour de la semaine où l'équipe commite le plus. La raison de ce pic est simple : à deux reprises lors de cette itération, nous nous sommes retrouvés ensemble physiquement, autour d'une grande table, pour travailler sur le projet.

Au début de la matinée, nous faisions une réunion pour établir les objectifs de la journée puis nous nous lancions tous sur la production de features. C'est lors de ces sprints que l'avancement était le plus marqué et que nous pouvions prendre du recul facilement sur ce qui avait déjà été fait, et ce qu'il restait à faire.

Développement itératif/fractal

Cette pratique a été adoptée dès le début, mais nous nous sommes rendus compte vers la moitié de l'itération que nous l'utilisions mal. Le principe de cette technique est de considérer chaque feature comme une unité qu'on peut développer à plusieurs niveaux de *perfection*.

Nous nous sommes forcés de produire très vite des proofs of concept, ou Minimum Viable Products pour les unités à développer. Cette unité était alors souvent à un stade de fonctionnalité très pauvre, dont l'architecture n'était pas forcément bien pensée. Il fallait alors réécrire, rajouter ou déplacer du code pour améliorer le point de vue utilité autant que le point de vue de beauté interne du code, afin d'arriver à une unité fonctionnelle complète et qui s'intégrait bien dans l'architecture de l'application.

Toutefois, lors de la première moitié de la première itération, nous sommes souvent tombés dans le piège du "*perfectionnement d'abord*", en passant beaucoup de temps sur le perfectionnement utilitaire de l'unité. Nous aurions dû passer plus vite - une fois que l'architecture d'une unité était bien intégrée dans l'application - au développement des autres unités.

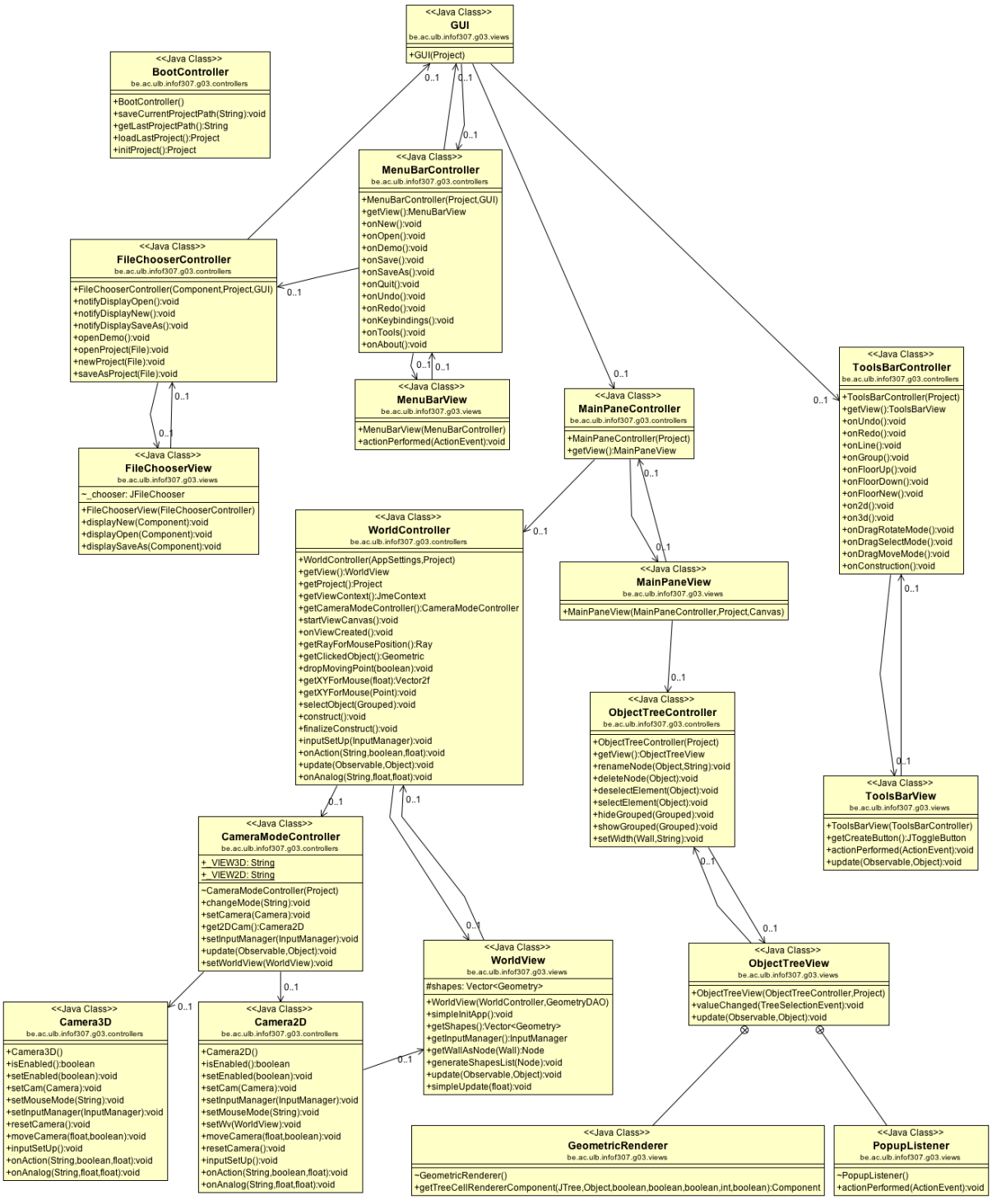

FIGURE 1.4 – Diagramme de classes de la GUI

Test-Driven Development

Cette technique, qu'il ne faut plus expliquer, a été utilisée, surtout pour le développement du modèle et s'est effectivement avérée positive lorsqu'il a fallu se baser sur un modèle robuste.

Staging area

Nous avons créé une branche `stage` sur laquelle nous testions la version de développement la plus récente de l'application. De manière régulière, nous passions tous les changements de `stage` en production sur `master`. La condition de mise en production était simple : tout le code doit être documenté, et tout ce qui peut être testé doit être testé.

Couverture de la qualité du code

Des outils d'évaluation de la qualité du code ont été utilisés. Parmi eux, on peut en noter trois :

EclEMMA Cet outil nous a permis d'évaluer facilement la couverture des tests dans l'application et nous permet d'estimer visuellement très rapidement cette dernière.

Missing Javadoc (Eclipse) On peut configurer Eclipse pour afficher des warnings là où le code n'est pas documenté, ce qui est très pratique.

PMD Plusieurs d'entre nous ont installé PMD en fin d'itération pour évaluer le respect des conventions de code. Nous n'avons cependant pas pris le temps de "réparer" les quelques erreurs mises en évidence.

Continuous integration

Nous avons choisi d'utiliser Travis CI [2] pour tester tous les commits pushés sur le dépôt, et ainsi assurer l'intégrité de la compilation et des tests, et d'être notifiés des erreurs éventuelles. Ant [3] nous permet de compiler le projet hors d'Eclipse.

Réunions hebdomadaires

Nous avons réussi à tenir notre objectif d'une réunion hebdomadaire. Le bilan de ces réunions est positif ; elles nous aidaient à prendre du recul sur notre progression et à remettre tout au clair.

1.4.4 Réflexion sur les librairies choisies

Généralement, nous sommes satisfaits de nos choix.

2. `http://travis-ci.org`
3. `http://ant.apache.org/`

jMonkeyEngine

Même si elle est plutôt facile à utiliser, nous nous sommes rendus compte que la documentation et les ressources disponibles pouvaient parfois être rares. Une grande partie de ce que nous avons trouvé venait directement du site officiel de jMonkeyEngine qui, en soi, est très complet, mais qui est parfois lacunaire sur certains sujets.

En prenant un peu de recul, nous nous rendons compte que nous utilisons des fonctionnalités d'assez bas niveau et que nous aurions pu nous satisfaire de moins.

Swing

Swing reste un excellent choix, nous n'avons pas à nous plaindre.

1.4.5 Conclusion - What's next

La fonctionnalité atteinte lors de cette itération est satisfaisante. Il en est de même pour la stabilité de la release. Il reste toutefois quelques légers problèmes d'expérience utilisateur, et quelques bugs mineurs persistent.

Nous avons déjà prévu de commencer l'itération par une revue du code et par un refactoring en tout cas du modèle. Nous avions par exemple fait le choix de pouvoir imbriquer des groupes dans des groupes (structure récursive), ce qui implique une trop grande complexité dans le `GeometryDAO`, n'est pas en adéquation avec les principes d'une base de données relationnelle, et n'est pas utilisé.

Chapitre 2

Itération 2

2.1 Introduction

Cette itération a vu prendre place le développement de l'histoire 2 proposée par le client : rajout des textures et création d'objets. Ce choix a été fait pour rendre l'application plus complète, et permettre plus de liberté artistique aux utilisateurs.

2.2 Histoires utilisateur

2.2.1 Vue Création d'objets

Durée estimée : 4h | Difficulté estimée : 1/3 | Assigné à : Pierre et Julian | Groupe **3**

Il faut rajouter une autre vue 3D que celle de l'éditeur principal, dans laquelle on peut créer des objets, et les modifier.

Statut : fait | Remarques :

2.2.2 Vue Libraire d'objets dans l'éditeur principal

Durée estimée : 7h | Difficulté estimée : 2/3 | Assigné à : Julian et Walter | Groupe **3**

Il est nécessaire de pouvoir rajouter des objets que l'utilisateur a créé dans l'éditeur général. Dès lors, il faut rajouter une vue qui permet de consulter tous les objets créés et d'en sélectionner un dont on peut créer une instance dans l'éditeur général.

Statut : fait | Remarques :

2.2.3 Librairie de textures

Durée estimée : 30h | Difficulté estimée : 2/3 | Assigné à : Pierre, Walter | Groupe **3**

Autant dans l'éditeur général que dans l'éditeur d'objets, on doit avoir accès à la librairie de texture qui se matérialise sous la forme d'un panneau de l'interface graphique dans lequel on peut avoir un aperçu des textures déjà importées dans le projet.

Ce panneau doit également donner la possibilité de rajouter une texture au projet, ou d'en supprimer. Ces changements seront réfléchis sur les objets qui portent ces textures.

On doit pouvoir rajouter une texture à un objet.

Statut : fait | Remarques :

2.2.4 Création d'objets et modification de leurs propriétés

Durée estimée : 40h | Difficulté estimée : 2/3 | Assigné à : Titouan, Bruno, Florentin | Groupe **3**

Dans la vue de création d'objets, on doit pouvoir rajouter des formes de base de jMonkey (parallélépipède rectangle, ellipsoïde, plan, ...). On peut changer la positition de ces formes, leur échelle sur les 3 axes et leur rotation, aussi sur les 3 axes. Chacune des formes peut porter une texture.

Statut : fait | La modification des objets n'est pas tout à fait user-friendly. Pour modifier un objet, il faut pour le moment modifier les composants de l'objet un par un.

2.3 Rapport de fin d'itération

La physionomie de l'itération 2 a été intrinsèquement différente de celle de son antérieure. Nous n'avons eu la confirmation que nous pouvions développer l'histoire 2 que 8 jours avant la fin de l'itération 2, ce qui a entraîné un rush plutôt conséquent. Toutefois, Nous y avions bien entendu déjà réfléchi et profité du début de l'itération pour entreprendre quelques refactorings rafraîchissants.

Anecdotiquement, on peut noter que c'est la troisième fois qu'un des membres de l'équipe est malade et ne peut participer activement au développement pendant plusieurs jours. La communication du groupe est restée forte et ces accrocs n'ont jamais eu d'impact ; nous avons toujours réussi à rééquilibrer les responsabilités sur les épaules de tous les membres.

2.3.1 Suivi du planning

À nouveau, pour cette itération, nous avons réussi à implémenter toutes les fonctionnalités que nous nous étions imposées, malgré le timeframe très court.

Nous avons gardé globalement la même architecture pour le projet, malgré quelques changements ça et là qui sont détaillés par la suite.

2.3.2 Architecture

Changements dans le modèle

Durant cette seconde itération, nous avons opéré un refactoring de notre modèle, suite à plusieurs problèmes que nous avions rencontré durant la première itération. Nous avions initialement prévu de pouvoir imbriquer récursivement les structures, de façon à créer des groupes de pièces par exemple. Cette structure, bien que très modulaire, est cependant inadaptée à une structure de base de donnée relationnelle, puisqu'elle implique d'effectuer une requête pour chaque type d'élément à chaque niveau de la récursion, et demande beaucoup de vérifications de types, contraires aux principes du polymorphisme.

Nous l'avons remplacé par un modèle à la hiérarchie fixe : un étage comprend des pièces, sur lesquelles on peut construire un mur, un sol et un plafond.

Les objets qu'on peut placer dans la pièce sont construits à partir de primitives (cube, sphere, pyramide et cylindre), pour créer des entités. On peut ensuite instancier ces entités dans le bâtiment.

Changements dans la GUI

CameraContext Le pattern state permet de gérer l'état des contrôleurs 2D et 3D de la caméra. Nous n'avons donc pas notre propre classe caméra, nous utilisons directement celle créée de la classe SimpleApplication. Afin de géré l'état, il nous a donc été nécessaire de créer une classe supplémentaire CameraContext. Cette classe sert de listener de tout les événement liés au interaction avec la caméra.

Lancement des contrôleurs Nous avons légèrement amélioré la manière dont nous lançons un contrôleur. Ils possèdent désormais une méthode `run()` qu'il faut appeler après leur construction ; cela nous permet de les lancer sans instancier de vue (pratique pour les tests, améliore le découplage). De plus, si quelque chose se passe mal dans cette fonction, on reste sûr que l'objet contrôleur a été bien construit, puisque son constructeur s'est terminé bien avant.

Rajouts dans la GUI

Texture Panel Il a fallu rajouter un panneau pour afficher les textures disponibles et offrir une interface à l'utilisateur pour en rajouter. Grâce à l'architecture MVC que nous avions mise en place lors de la précédente itération, c'était très facile et il a suffi de rajouter une TextureView et un TextureController.

Création d'objets Pour l'instant nous avons ajouté tout ce qui permet de manipuler les objets dans les contrôleurs existants. Les vues tel que la treeView et le canvas jmonkey sont réutilisé pour l'édition/affichage des objets. Par manque de temps nous n'avons pas pu fournir de solution architecturalement plus propre, cependant nous sommes conscient que le design pattern state pourrait largement améliorer et simplifier notre architecture. Nous avons comme projet de l'implémenter au plus vite avant de commencer la prochaine itération.

2.3.3 Bonnes pratiques - Outils

Nous avons continué à utiliser toutes les bonnes pratiques listées dans le rapport de l'itération précédente. Nous avions toutefois omis de préciser que nous utilisons **Trello** pour nous aider à identifier, assigner, séparer et préciser les tâches qui nous attendent.

2.3.4 Conclusion - What's next

Malgré une satisfaction générale de l'atteinte de nos objectifs, il nous reste un goût de trop peu à la fin de cette itération. Il reste quelques points que nous aurions aimé perfectionner.

Documentation et tests La documentation couvre une bonne partie du code mais est dans certains cas lacunaire ou inexistante. Certaines parties du code restent aussi non-testées.

Usabilité Certaines des actions que l'utilisateur peut faire dans l'application sont quelque peu obscures. Cela est dû au rajout incrémental de fonctionnalités, qui empêche de prévoir facilement ce qu'il sera possible de faire par la suite, et qui requiert parfois de repenser entièrement l'interface. Nous n'avons pas eu le temps de le faire lors de cette itération.

Responsabilités du GeometryDAO Comme expliqué dans les remarques fournies, le GeometryDAO a trop de responsabilités et doit être revu. Cela sera fait pour la prochaine itération.

Chapitre 3

Itération 3

3.1 Introduction

L'équipe a choisi d'implémenter l'importation et l'exportation d'objets en différents formats : OBJ, DAE (Collada), KMZ et 3DS. Le projet se rapproche de plus en plus à un logiciel d'architecture très complet permettant de visualiser assez rapidement une maison en cours de création.

3.2 Répartition des tâches

Lors de cette itération, nous n'avons pas séparé l'histoire utilisateur principale en sous-histoire pour une bonne et simple raison : nous voulions davantage passer d'une tâche à l'autre pour encore plus nous familiariser avec l'entièreté du projet.

Toutefois, comme l'indiquent les PV des réunions de cette itération, voici la manière dont les responsabilités ont été réparties au départ du développement :

- refactoring DAO : Titouan
- interface graphique import/export : Pierre
- découverte générale de tous les formats : Bruno
- refactoring textures : Walter
- refactoring State pour WorldController : Julian

Une fois que toutes ces tâches relativement petites ont été exécutées, tous les membres du projet se sont attelés au développement des parsers et exporteurs. Nous avons beaucoup utilisé le principe du pair programming, en sautant d'un format à l'autre, ce qui s'est avéré être une excellente technique de travail.

3.3 Rapport de fin d'itération

Nous avons rencontré moins d'accrocs lors de cette phase que lors des deux premières. Aucun événement extérieur indépendant de notre volonté nous a bloqué, et nous avions toutes les cartes en main dès le début ; cette itération fut donc une promenade de santé sur un long fleuve tranquille sans cascade ni haine ni violence dans la paix et l'entente de groupe les plus totales.

3.3.1 Follow-up sur le rapport de fin d'itération précédente

Dans le rapport de fin de l'itération précédente, nous mettions le doigt sur quelques problèmes qui nous ont été rappelés par la suite par les assistants.

Responsabilités du GeometryDAO (et d'autres classes)

Le `GeometryDAO`, classe centrale du projet, a grandi de manière indésirable et presqu'incontrôlée jusqu'à l'itération 2. Son architecture nous forçait à écrire beaucoup de méthodes spécifiques ou de tests de type à beaucoup d'endroits du DAO. Nous avons réglé ce problème en créant une Factory de DAO qui sera expliquée plus bas dans la section Architecture.

Le WorldController a aussi été identifié comme étant une classe trop grande et avec trop de responsabilités hétérogènes. Ce problème a été taclé grâce à l'implémentation d'un pattern State, lui aussi décrit plus loin dans ce rapport.

Documentation et tests

La documentation a été en grande partie complétée lors de cette itération. Pour ce qui est des tests, nous utilisons de plus en plus le *test-driven development* ; notamment dans l'écriture des Parsers pour laquelle nous écrivions très vite des tests pour fixer leur comportement, et aussi du *regression testing* : nous écrivions un test à chaque fois que nous nous heurtions à un comportement inattendu.

Meilleure répartition des packages

Il nous a été conseillé de répartir les classes par fonctionnalité plutôt que par "rôle" dans le pattern MVC. Cela a été fait.

3.3.2 Suivi du planning

Encore une fois, toute les fonctionnalités prévues ont été implémentées. Nous sommes également plutôt heureux d'avoir pu effectuer assez vite quelques refactorings importants. Bien que nous n'ayons pas peur de refactorer en plein développement, la période de début d'itération s'avère être un excellent moment pour remettre tout le projet au clair.

3.3.3 Architecture

Rajouts dans le modèle

Rajout des parsers Nous avons créé un `ImportEngine` qui va gérer les demandes d'import de l'utilisateur et la création/gestion des différents parsers à sa disposition.

Chaque parser hérite de la classe `Parser` qui impose un comportement standard pour chacun d'entre eux. L'`ImportEngine` travaille dès lors très facilement avec des Parsers qui ont un comportement générique. Il faut donc très peu le modifier si on désire rajouter

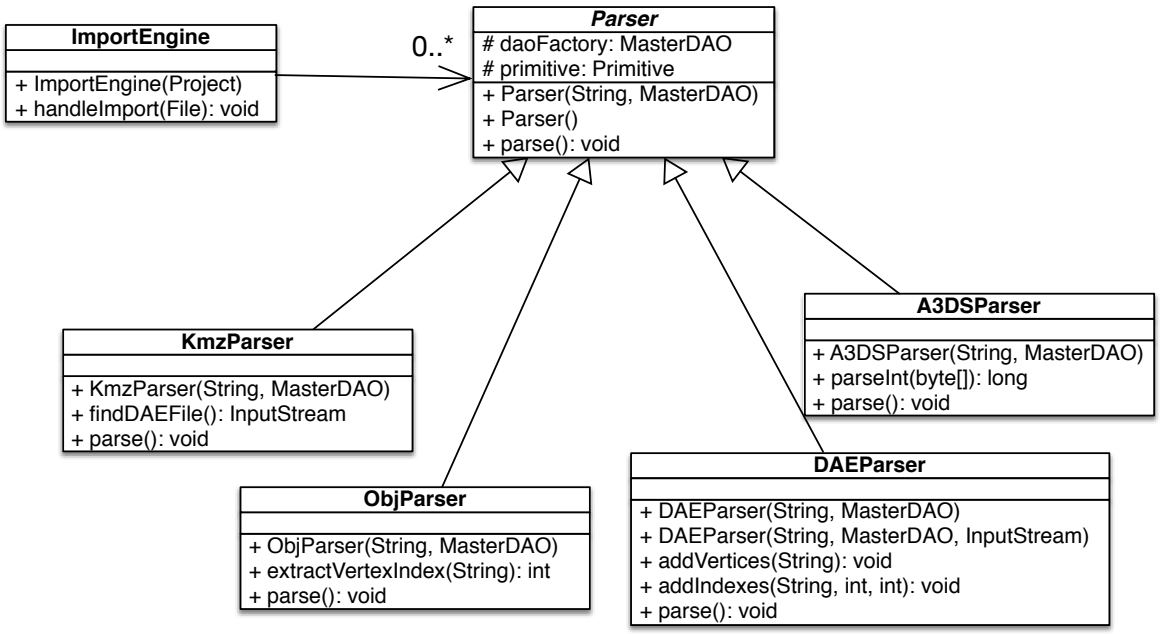

FIGURE 3.1 – Architecture du système d'import

un Parser.

L'architecture du système d'importation est visible à la figure 3.1.

Rajout des exporters À l'image de l'`ImportEngine`, nous avons aussi créé un `ExportEngine` qui gère les demandes d'export de l'utilisateur et sa collection d'exporters.

Primitives compatibles avec l'import/export Le modèle supporte maintenant l'ajout de primitives non standard dans un Item qui sont désormais composées de Vertex et d'Indices (déterminant les faces) afin de faciliter l'import/export d'objets qui ne sont pas des primitives de jMonkeyEngine.

Changements dans le modèle

Refactoring du GeometryDAO La figure 3.2 montre le diagramme de l'ancienne class GeometryDAO, et la figure 3.3 montre l'interaction que les vues et les contrôleurs avaient avec le DAO. On voit très clairement que l'ancien DAO dont la figure prend une page A4 en hauteur avait trop de responsabilités et avait des méthodes beaucoup trop spécifiques.

Nous avons donc changé le GeometryDAO en MasterDAO qui est une Factory de GeometricDAO génériques. Le MasterDAO entretient donc un GeometricDAO pour chaque type d'objet à insérer dans la base de données. Cette nouvelle architecture est visible en figure 3.4. Les nouvelles interactions des vues et des contrôleurs avec les DAO sont

représentées en figure 3.5.

Le lien du GeometricDAO vers le MasterDAO nous permet de transférer les changements pour un certain objet du modèle vers le MasterDAO, qui est le seul objet que toutes les vues écoutent. On garde donc un seul lien vers le modèle dans les contrôleurs et les vues tout en découplant très fort son accès, et en rendant très facile le rajout de différents types d'objets.

Prenons un exemple d'exécution en nous aidant de la figure 3.5. Si un contrôleur veut changer un objet `Floor`, il demande au MasterDAO le `GeometricDao<Floor>`. Le contrôleur appelle la méthode `modify()` à laquelle il passe le `Floor` en question en paramètre. Le `GeometricDao<Floor>` prend en charge l'interaction avec la base de données, puis transfère le changement au MasterDAO ; qui notifie tous les objets qui l'écoutent : toutes les vues sont donc mises au courant de la mise à jour.

Changements dans la GUI

Refactoring du WorldController Au terme de la deuxième itération, le WorldController prenait en charge la gestion du *monde* (bâtiment, etc) et la gestion des objets. Ces 2 comportements n'étant pas liés logiquement et pouvant être séparés, tout en devant garder la même vue (le canevas jMonkey), nous avons séparé le WorldController en plusieurs classes grâce au pattern State. On peut voir un schéma minimaliste de cette nouvelle architecture en figure 3.6.

La WorldView a un contrôleur différent en fonction du mode d'édition dans lequel le programme se situe.

Rajouts dans la GUI

Import/Export Nous avons rajouté des menus d'import et d'export pour les objets.

3.3.4 Bonnes pratiques - Outils

3.3.5 Conclusion - What's next

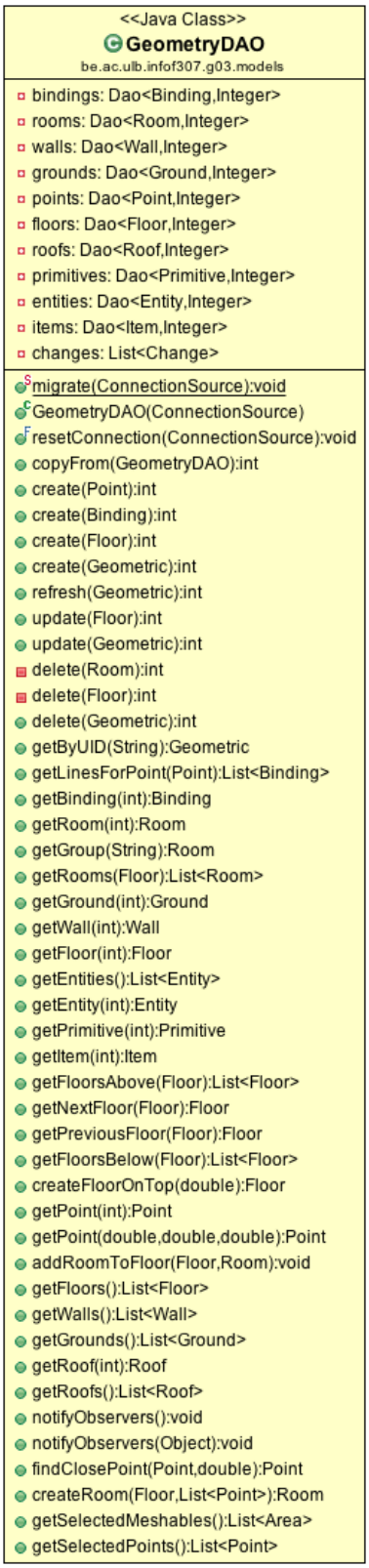

FIGURE 3.2 – Vieux GeometryDAO

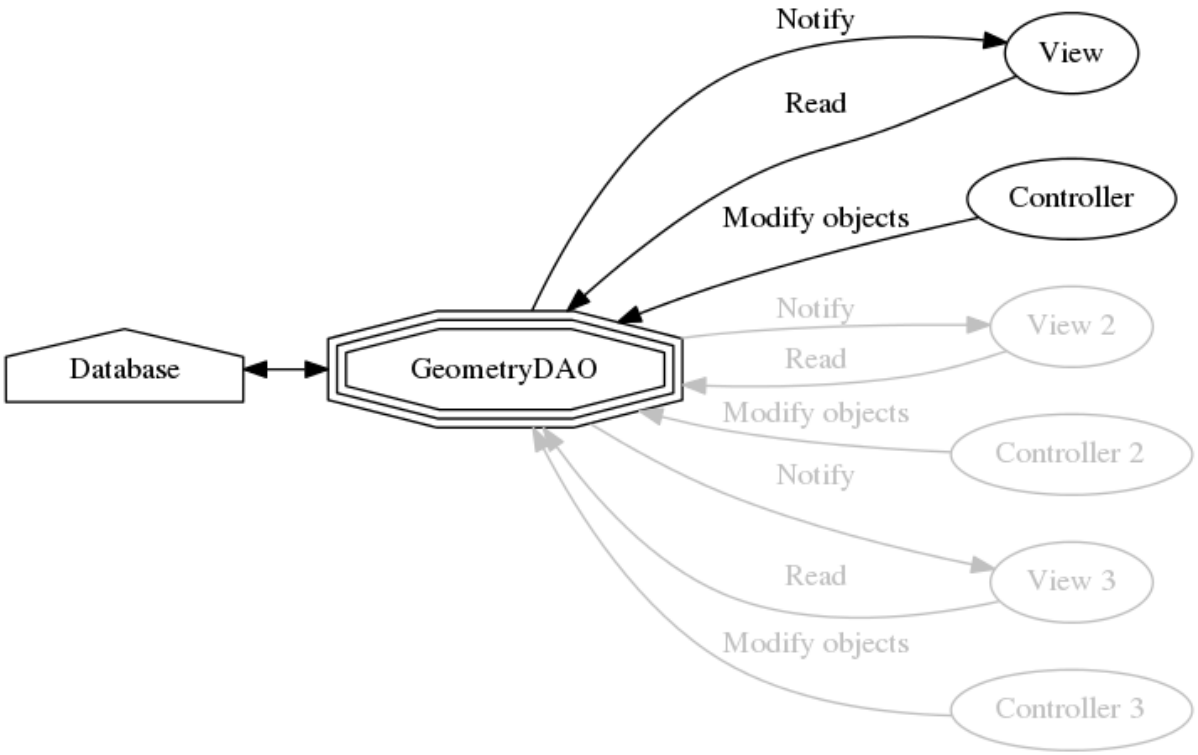

FIGURE 3.3 – Vieilles interactions avec le DAO

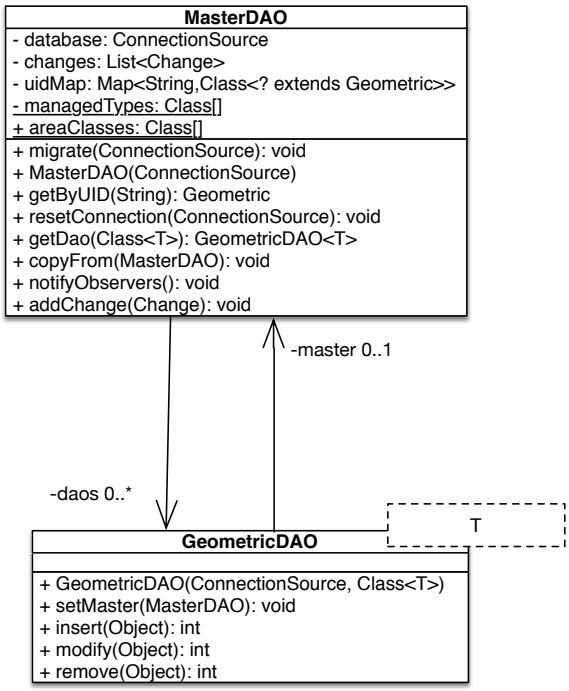

FIGURE 3.4 – Nouvelle architecture des DAO

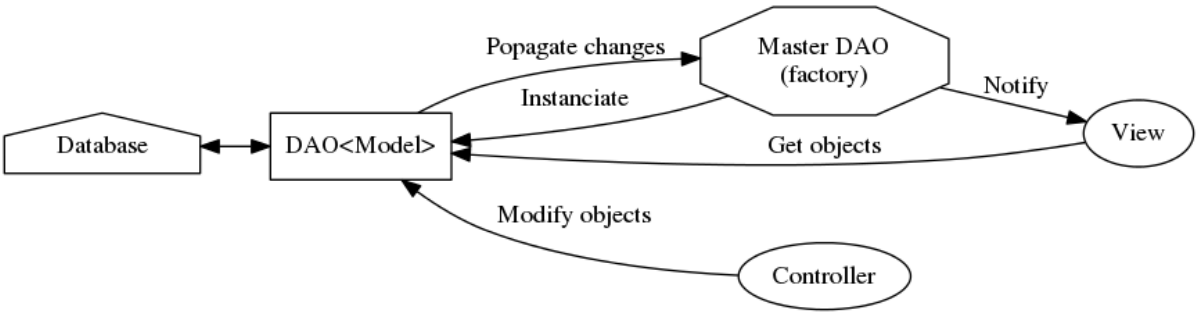

FIGURE 3.5 – Nouvelles interactions avec les DAO

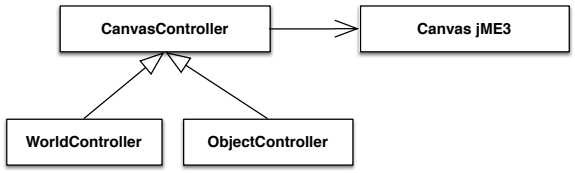

FIGURE 3.6 – Architecture des contrôleurs de la WorldView