Rapport du jalon 1 concernant l'implémentation de la couche « métier » du projet.

Université de Bourgogne
UFR Sciences et techniques
Licence 3 - Informatique

Rapport CDAA

Jalon 1

Clément GILI – Eddy DRUET – Groupe TP 02

TABLE DES MATIERES

Conception « UML » 2
Spécifications des classes 3
Classe « Date » 3
Classe « AppState » 3
Classe « ContactModel » 3
Classe « InteractionModel » 3
Classe « TodoModel » 3
Classe « Collection » 3
Identifiant des modèles 4
Relations entre les modèles 4
Contraintes sur les données 4
Documentation Doxygen 5
Tests unitaires, main 5
Organisation du travail 5

CONCEPTION « UML »

Tout d'abord, avant de continuer sur l'explication et la justification de la conception du projet, ci-dessous un diagramme de classes de la couche « métier » pas exactement aux normes UML, mais qui donne un bon aperçu de l'implémentation de l'architecture et la relation entre les classes.

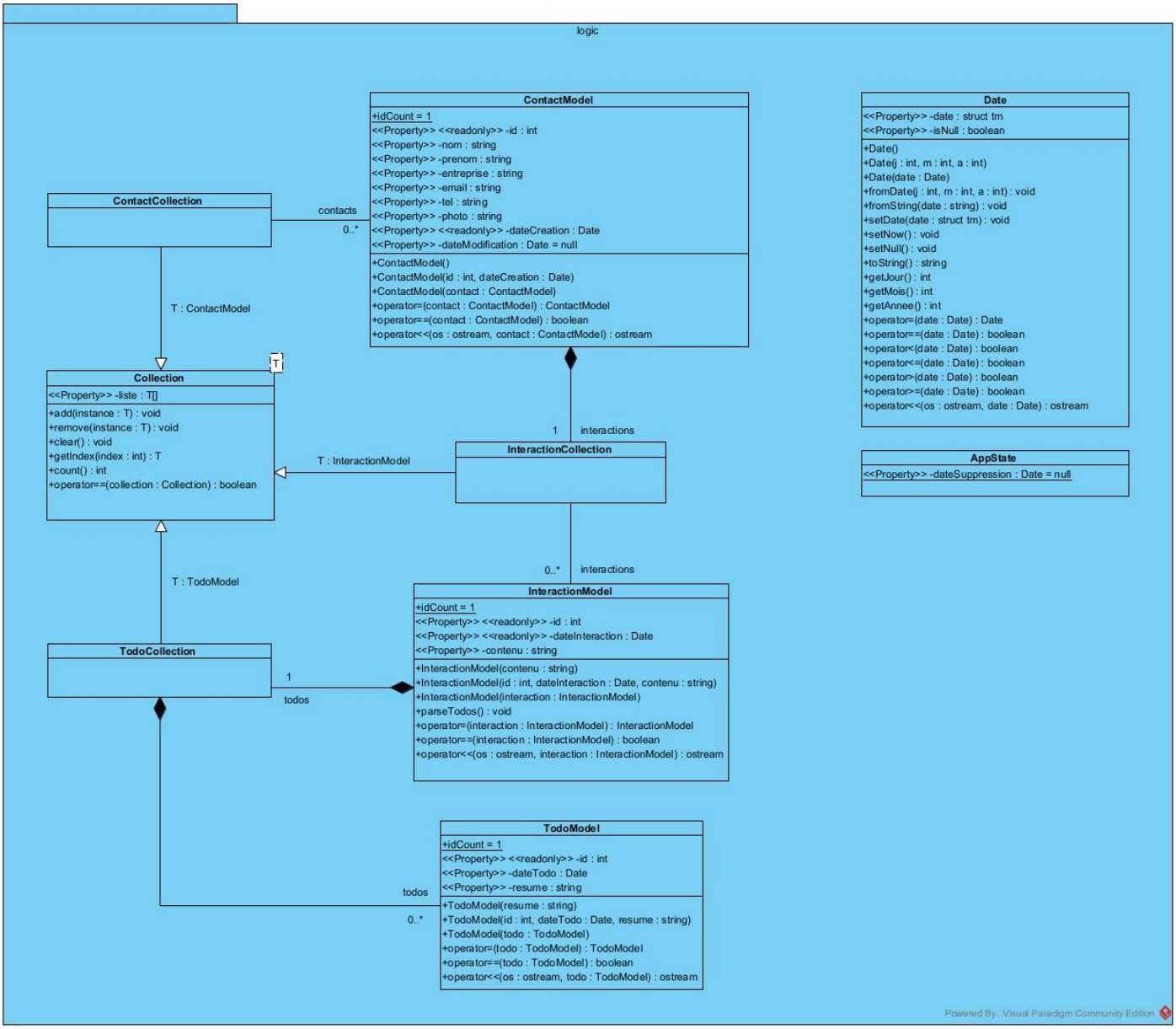

Figure 1 : Diagramme de classes de la couche « métier »

Le projet a été effectué de sorte à essayer de respecter au maximum les principes de la programmation orientée objet (encapsulation, héritage, généricité, etc.), mais aussi des pratiques et conventions du langage C++, comme le « const-correctness », la surcharge d'opérateurs et de constructeurs, les itérateurs, l'utilisation du moins possibles des pointeurs, etc.

SPECIFICATIONS DES CLASSES

Dans cette partie, sont expliqués le but, la structure, et les particularités de chaque classe.

CLASSE « DATE »

Le stockage et la manipulation des dates en C++ standard étant peu pratique, la classe « Date » encapsule une date C et permet de simplifier la manipulation en offrant une interface pratique. La classe ne gère pas le temps, car, à priori, ce ne sera pas nécessaire dans le projet, mais permet de gérer les dates nulles. Les opérateurs de comparaisons sont surchargés pour effectuer des vérifications et contraintes sur les données. La classe permet également de définir une date par plusieurs moyens : à partir d'un string, d'un jour/mois/année, de la date du jour et peut convertir une date en string.

CLASSE « APPSTATE »

Cette classe représente et stocke les états globaux de l'application tels que la dernière date de suppression d'un contact. Ces états sont déclarés de façon « statique » pour qu'ils puissent être accessibles à tout endroit du projet car ils ne sont pas spécifiques à une instance.

La date de la dernière suppression est mise à jour lorsqu'un contact, peu importe lequel, est supprimé. Ceci sera effectué lors de la confirmation de la suppression (ex. quand on clique sur le bouton « Supprimer ») dans l'interface utilisateur de l'application Qt.

CLASSE « CONTACTMODEL »

Ce modèle stocke les informations d'un contact. Un contact possède une liste d'interactions qui lui sont propres, un contact a donc une relation forte avec ses interactions.

La date de création est par défaut initialisée à celle du jour, et la date de dernière modification est par défaut nulle puisqu'il n'y a pas eu encore de modification. Tous les attributs peuvent être vides sauf « id », et les dates, puisqu'on suppose que le contact peut ne pas renseigner toutes ses coordonnées.

Les dates de modification seront modifiées uniquement au moment de la confirmation de la modification (ex. quand on clique sur le bouton « Modifier » d'une fiche contact) dans l'interface utilisateur de l'application Qt. La raison est que définir un attribut n'est pas signe de modification, cela peut être simplement être une première initialisation (ex. suite au chargement des données depuis la base de données).

CLASSE « INTERACTIONMODEL »

Ce modèle stocke les informations d'une interaction d'un contact. Comme la classe « ContactModel », cette classe entretient une relation forte avec les todos, puisqu'elle possède une liste de todos issus de l'interaction après analyse de son contenu.

Le contenu de l'interaction, comme le sujet le précise, peut contenir des commentaires et des tags comme « @todo » et « @date ». La méthode « InteractionModel::parseTodos() » permet de démarrer l'analyse du contenu de l'interaction, pour y extraire les todos ainsi qu'éventuellement leur date de réalisation (si aucune n'est précisée, alors celle-ci est la date du jour). Cette extraction conduit à l'instanciation de « TodoModel » et à l'ajout dans la liste de todos de l'interaction. La création des todos se fait donc dans cette classe suite à l'appel de la méthode précédente.

CLASSE « TODOMODEL »

Ce modèle stocke les informations d'un todo d'une interaction. La classe est basique, il n'y a pas de choses particulières concernant celle-ci.

CLASSE « COLLECTION »

Cette classe est une classe générique permettant de stocker et manipuler une liste d'instances d'un type particulier T. La classe encapsule une liste C++ standard.

Les classes « ContactCollection », « InteractionCollection » et « TodoCollection » héritent de cette classe et se spécialisent pour leur modèle respectif. L'intérêt de ces classes collections par rapport à une liste C++ standard, est de rajouter une couche d'abstraction pour y ajouter des opérations communes ou spécifiques à des collection de modèles à l'avenir, notamment pour le système de requêtes du prochain jalon. Ces opérations seront par exemple : trier ou filtrer la liste selon un attribut spécifique.

IDENTIFIANT DES MODELES

Dans chaque classe modèle, se trouve un attribut public et statique « idCount ». Celui-ci est un compteur démarrant à 1 et qui s'incrémente à chaque nouvelle instanciation d'un modèle par le premier constructeur. Il permet de garder trace du dernier identifiant généré pour définir l'identifiant de la prochaine instance. Cela permet de garder l'unicité des identifiants pour le stockage dans la base de données par la suite. Cet attribut sera par la suite initialisé par l'identifiant maximal se trouvant dans la base de données pour reprendre où en était le compteur.

Pour chaque modèle, il existe un constructeur pour créer une instance vierge (identifiant généré automatiquement) et un autre constructeur pour pré-initialiser l'instance (identifiant prédéfini) pour charger une instance avec les données récupérées de la base de données.

RELATIONS ENTRE LES MODELES

La structure choisit reliant les classes entre-elles est une structure hiérarchique entre les entités contact-interaction-todo. Celle-ci permet de faciliter la gestion du parcours et de garder une complexité réduite de la gestion des relations entre les classes (ex. pas de relations bidirectionnelles). L'inconvénient de ce choix étant la perte de performance concernant la recherche, notamment dans le cas où l'on veut effectuer une recherche globale sur les interactions ou les todos sans prise en compte des contacts, où il faudra « inutilement » itérer sur tous les contacts pour accéder à toutes les interactions et todos.

CONTRAINTES SUR LES DONNEES

Un certain nombre de contraintes sur les données ont été implémentées. Lorsqu'une valeur d'un attribut est définie et qui viole une contrainte, une exception est déclenchée avec un message expliquant la violation de contrainte. Cela permet de garder une cohérence des données dans les modèles, et permettra par la suite dans l'application Qt, de gérer ses exceptions par une dialogue montré à l'utilisateur s'il inscrit une donnée invalide.

Classe	Attribut	Violation de contrainte
ContactModel	dateCreation	Date de création vide
	dateModification	Nouvelle date de modification < date de création Nouvelle date de modification < date de modification actuelle
	email	Format adresse e-mail invalide
	tel	Format numéro de téléphone invalide
InteractionModel	dateInteraction	Date vide
	contenu	Contenu vide
TodoModel	resume	Résumé vide
	dateTodo	Date vide
AppState	dateSuppression	Nouvelle date de suppression < date de suppression actuelle
Date	fromDate()	Jour, mois, année non valide
	fromString()	Format de date invalide

Tableau 1 : Liste des contraintes sur les données

DOCUMENTATION DOXYGEN

Tout le projet a été documenté avec « Doxygen » de façon extensive, la documentation a été générée au format HTML.

TESTS UNITAIRES, MAIN

Toutes les méthodes de toutes les classes ont été testées avec des tests unitaires pour vérifier le bon comportement, le respect des spécifications et des contraintes (vérification du bon déclenchement des exceptions). Un certain nombre de bugs ont été décelés et corrigés grâce aux tests unitaires réalisés.

Le « main » exécute tous ces tests unitaires, et en dernier, réalise un test concret global de la couche : création d'un contact, assignation d'interaction et création de todos avec affichage des étapes dans la console.

ORGANISATION DU TRAVAIL

Nous nous sommes divisé l'implémentation des classes, chacun devant implémenter une classe modèle, ainsi que la moitié des tests unitaires. L'objectif étant que chacun pratique la même chose et apprenne autant sur le langage.