Rapport final du projet de CDAA expliquant l'architecture du projet, les fonctionnalités implémentées, les choix et les instructions d'utilisation et de compilation.

Université de Bourgogne
UFR Sciences et techniques
Licence 3 - Informatique

Rapport CDAA

Rapport final

Eddy DRUET – Clément GILI – Groupe TP 02

TABLE DES MATIERES

INTRODUCTION

Ce projet du module de « Conception et développement d'application avancée » nous demandait de développer une application Qt en lien avec une base de données SQLite. Cette application permet de gérer une liste de contacts, où il est possible d'en créer, modifier et supprimer. Par ailleurs, l'application permet de gérer des interactions à des contacts, ainsi que des todos à réaliser pour une certaine date. Il est possible d'effectuer une recherche de contacts selon différents critères, mais aussi de faire des requêtes sur les interactions et les todos. Enfin, l'application finale enregistre les données dans la base de données, et offre la possibilité d'exporter au format JSON, l'ensemble des données.

Nous allons, dans ce rapport, expliquer les fonctionnalités implémentées, l'architecture de l'application, ainsi que les instructions de compilation et d'utilisation de l'application.

FONCTIONNALITES IMPLEMENTEES

Toutes les fonctionnalités attendues du sujet ont été réalisées. Ci-après, vous allez voir chaque fenêtre de l'application et vous aurez une explication des fonctionnalités réalisées.

FENETRE PRINCIPALE

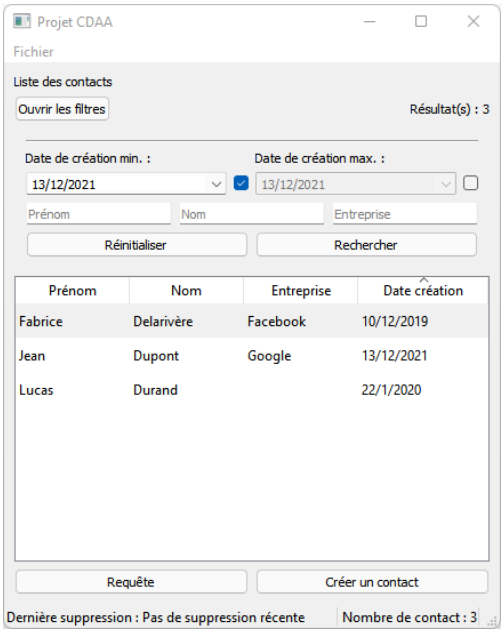


Image 1 : Fenêtre principale.

La fenêtre principale liste les contacts avec leur « prénom », « nom », « entreprise » et « date de création ». Il est possible de trier chaque attribut en cliquant dessus comme ici pour la « date de création ».

Il est possible d'ouvrir un panneau de recherche en cliquant sur « Ouvrir les filtres » pour rechercher des contacts. Si les deux dates sont cochées, cela recherche les contacts avec une date de création dans l'intervalle min-max. Vous pouvez passer la souris sur les cases pour avoir plus de détails.

La barre de statuts indique la date de la dernière suppression d'un contact et affiche le nombre de contacts total (hors recherche).

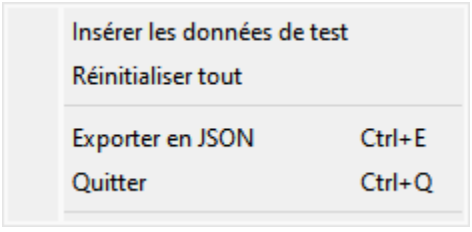


Image 2 : Menu de la fenêtre principale.

Le menu permet de :

- « Insérer les données de test » pour avoir quelques contacts, interactions et todos de test ;
- « Réinitialiser tout » pour supprimer toutes les données ;
- « Exporter en JSON » vers un fichier.

FENETRE DE REQUETE

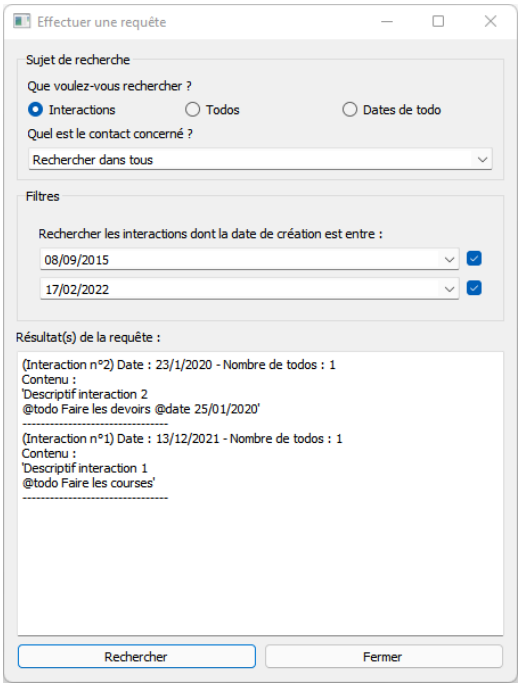


Image 3 : Fenêtre de requête.

La fenêtre de requête permet de faire des recherches sur les interactions, todos et dates de todo et d'afficher le résultat.

Il est possible de centrer la recherche sur tous les contacts, ou bien sur un contact en particulier.

Par ailleurs, il est possible de filtrer par la date de création. Comme pour la recherche de la fenêtre principale, cocher les deux cases permet de chercher les dates de création dans l'intervalle. Pour plus de détails, passez la souris sur les cases.

FENETRE D'EDITION D'UN CONTACT

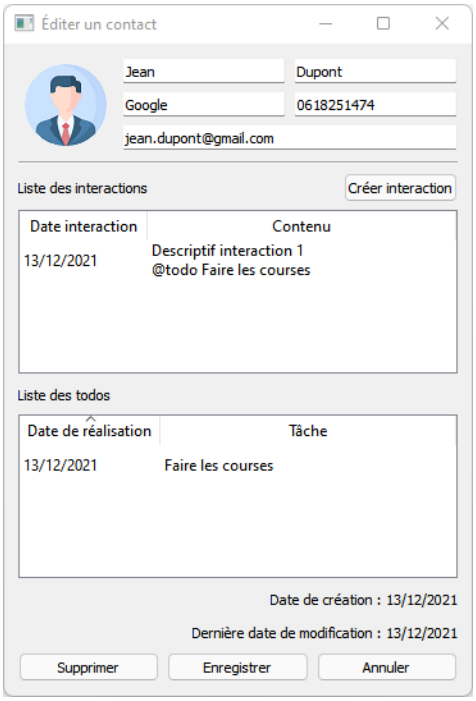


Image 4 : Fenêtre d'édition d'un contact.

La fenêtre d'édition d'un contact permet de créer ou de modifier un contact ainsi que leurs interactions et todos. Il n'est possible de créer des interactions que lors de la modification d'un contact.

La photo du contact est modifiable en cliquant directement sur l'image.

Comme pour la liste des contacts, il est possible de trier la liste d'interactions et de todos. En sachant, que par défaut, les todos les plus urgents sont affichés en premier.

FENETRE D'EDITION D'UNE INTERACTION

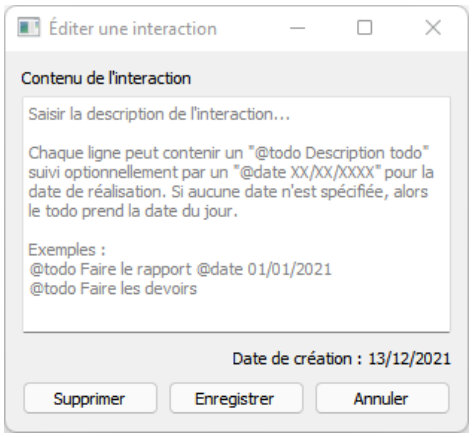


Image 5 : Fenêtre d'édition d'une interaction.

La fenêtre d'édition d'une interaction permet simplement de créer ou de modifier une interaction.

NOTES SUPPLEMENTAIRES

Toutes les fenêtres gèrent les erreurs qui peuvent survenir (ex. erreur lors des opérations dans la base de données, violation de contraintes sur les données, etc.) en affichant un dialogue d'erreur tout en annulant le processus qui était en cours (ex. modification d'un contact).

CONCEPTION « UML »

Nous allons, dans cette partie, voir la conception « UML » de l'architecture de l'application. Nous avons déjà vu l'architecture de la couche métier au jalon n°1, nous n'allons donc pas tout réexpliquer concernant cette partie, mais juste expliquer les ajouts et modifications apportés. Nous verrons également les autres couches qui ont été ajoutées.

DIAGRAMMES DE PACKAGES

L'architecture de l'application a été découpée en quatre grandes parties. Cela permet de bien séparer les différents niveaux de l'application, de réduire les dépendances et donc favoriser la réutilisation des couches dans d'autres projets en tant que bibliothèques. Les dépendances entre les couches étant bien maîtrisées, cela permet également de faciliter la maintenabilité du code et éviter les effets de bords.

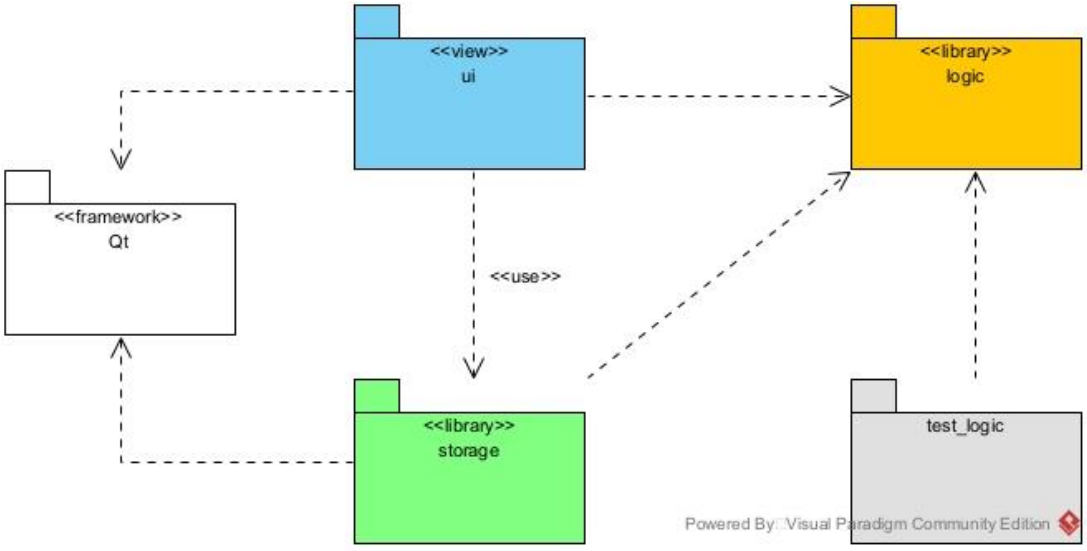


Diagramme 1 : Diagramme de packages de l'application.

PACKAGE « LOGIC »

Ce package représente la couche métier, c'est celle qui a été décrite lors du jalon n°1. Cette couche est le moteur de l'application et a pour but le stockage des données en mémoire, ainsi que des opérations de manipulation de ces données.

Cette couche est totalement indépendante des autres et ne contient aucun code de « haut niveau » comme l'interface utilisateur, ou la sérialisation des données. L'intérêt de cette indépendance, est qu'elle peut être intégrée à n'importe quel autre projet, même non Qt, puisqu'elle utilise le C++ standard.

PACKAGE « TEST LOGIC »

Ce package contient simplement les tests unitaires de la couche métier. Elle dépend donc directement de la couche métier.

PACKAGE « STORAGE »

Ce package représente la couche de stockage, c'est-à-dire, la persistance des données dans un support comme une base de données ou dans un fichier. Ainsi, cette couche est responsable de l'interaction directe avec notre base de données SQLite pour charger ou enregistrer les données. Cette couche dépend donc entièrement des données de la couche métier.

Par ailleurs, cette couche dépend fortement de Qt, puisqu'elle utilise ses modules et types pour gérer la persistance des données.

PACKAGE « UI »

Ce package représente la couche IHM. Elle dépend de Qt pour réaliser les fenêtres et l'interaction avec l'utilisateur. Elle dépend de la couche de stockage pour charger les données et enregistrer les données lors des actions de l'utilisateur. Par ailleurs, elle utilise la couche métier pour afficher les données en mémoire et les modifier.

DIAGRAMMES DE CLASSES

Maintenant, que nous avons vu l'organisation globale de l'architecture de l'application, nous allons voir plus en détail chacune des couches et décrire leurs liens, les rôles et l'implémentation des classes.

COUCHE METIER

Nous n'allons pas tout réexpliquer pour cette couche, si vous voulez plus d'informations, consultez le rapport du jalon n°1. Cependant, nous avons ajouté des fonctionnalités pour les classes « Collection ».

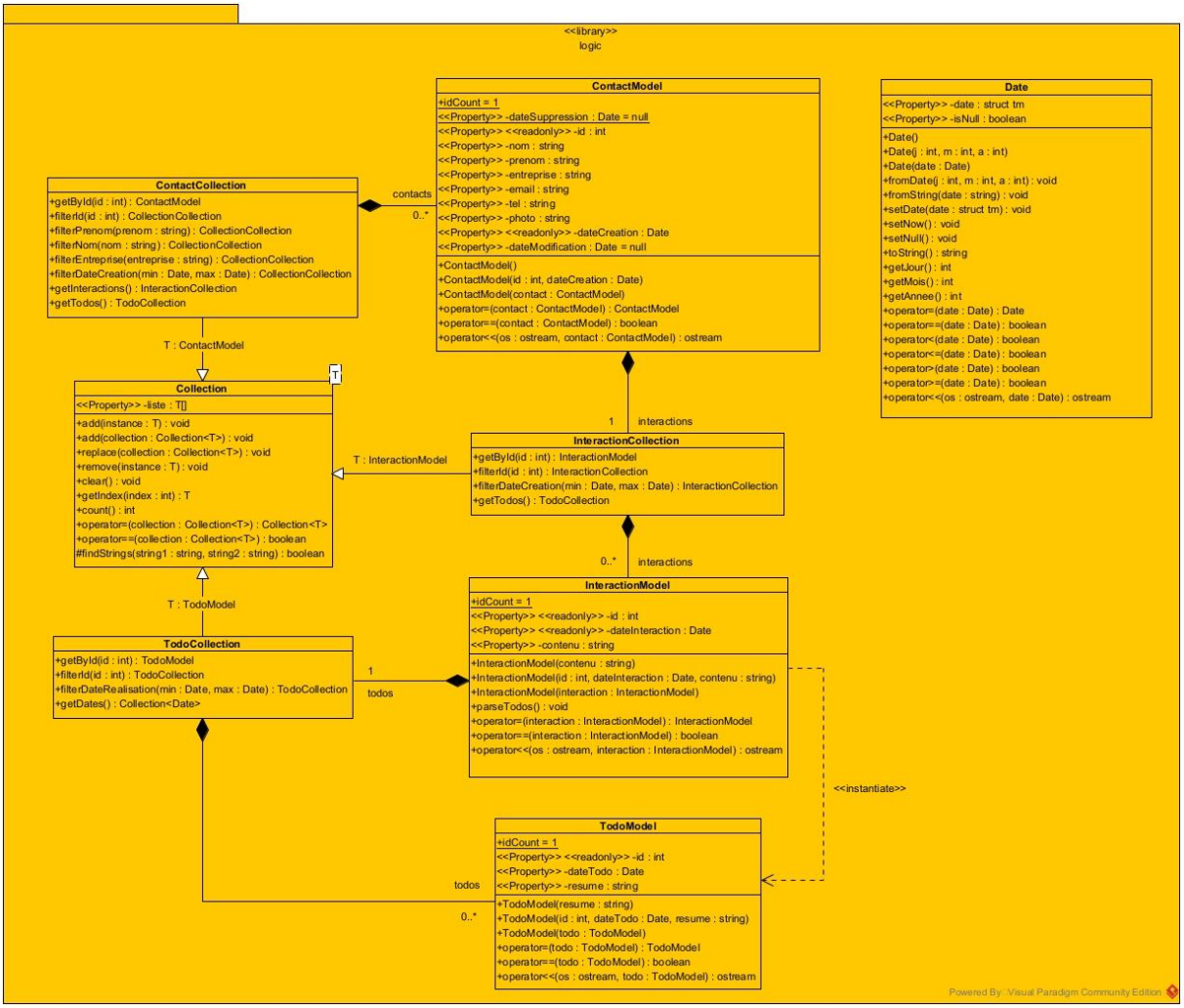


Diagramme 2 : Diagramme de classes de la couche métier.

Au départ, les classes « ContactCollection », « InteractionCollection » et « TodoCollection » ne contenaient aucune méthode propre à part celles héritées. Cependant, afin, d'effectuer le système de recherche de contacts et le système de requête demandés dans le sujet, nous avons ajouté des méthodes permettant de filtrer le contenu des collections.

Par exemple, pour la classe « ContactCollection », il est possible de filtrer tous les contacts de la collection appartenant à une certaine entreprise, ou bien tous les contacts nommés « Jean ». Ces méthodes de filtres ne prennent pas en compte la casse et recherchent la présence d'une chaîne de caractères dans l'attribut, ainsi, il est possible d'obtenir tous les « Jean » en cherchant « an ».

Ces classes implémentent le design pattern « Désignation chaînée » permettant de faire une chaîne d'appel de méthodes, par exemple « collection.filterNom("Jean").filterEntreprise("Google") ».

COUCHE DE STOCKAGE

Cette couche permet de gérer la persistance des données. Elle gère le chargement et l'enregistrement depuis la base de données SQLite, l'exportation des données au format JSON, ainsi que la gestion de la connexion à la base de données.

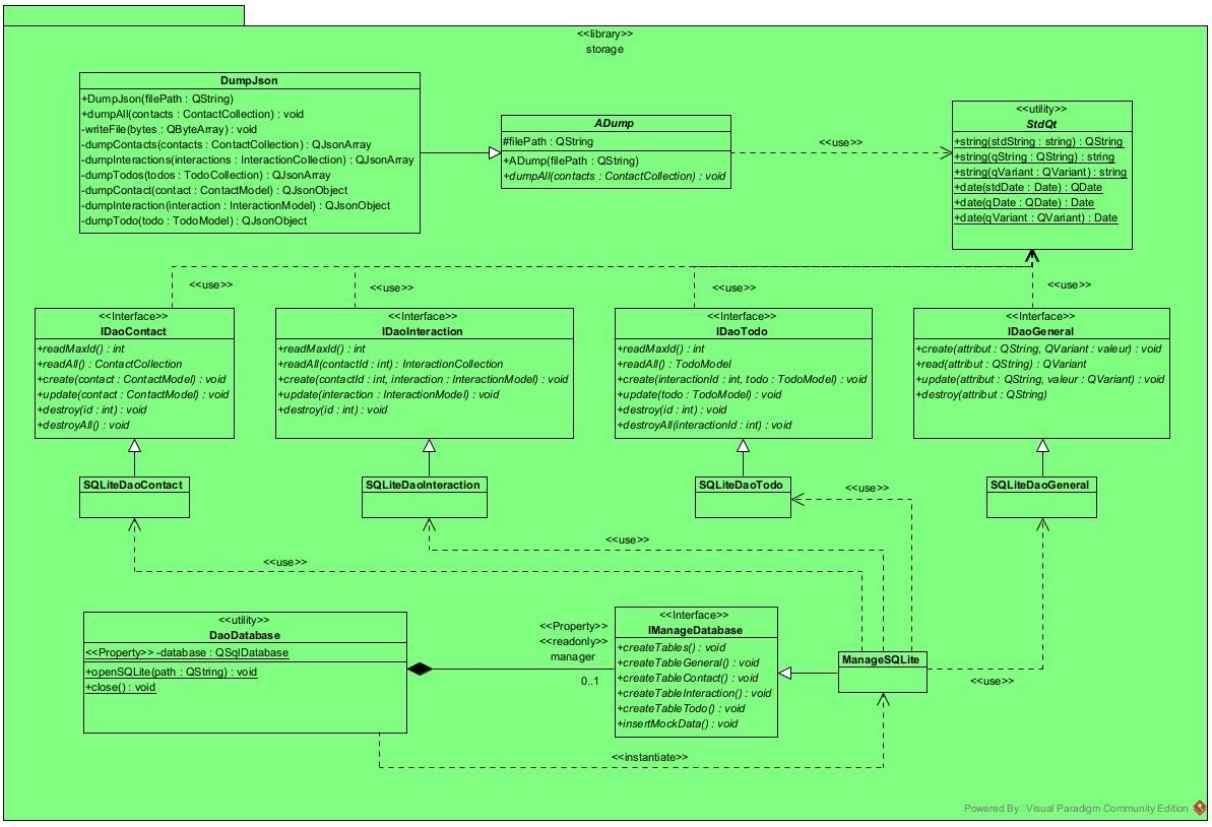


Diagramme 3 : Diagramme de classes de la couche de stockage.

CLASSE « DAODATABASE »

Cette classe permet de gérer la connectivité avec la base de données. Ici, il est possible d'ouvrir une base de données SQLite en lui donnant le chemin vers le fichier. Il est également possible de fermer la base de données. Elle possède un « manager » (propre à SQLite ici), lui permettant de créer la structure de la base de données si celle-ci se trouve vierge au moment de l'ouverture de la connexion.

Cette classe est statique, car la connexion à la base de données est globale dans toute l'application, c'est-à-dire, qu'une seule connexion est possible à la fois.

En cas de problème lors de la connexion, ou la création de la structure de la base de données, une exception est déclenchée et la base de données est effacée et recréée au prochain lancement de l'application.

INTERFACE « IMANAGEDATABASE » ET CLASSE « MANAGESQLITE »

Cette interface propose des méthodes permettant la création de la structure de la base de données (tables principalement). Chaque méthode permet de créer la table correspondante si celle-ci n'existe pas déjà. Par ailleurs, la méthode « insertMockData() » permet d'insérer des données de test (contacts, interactions et todos) pour le débogage. Par ailleurs, une interface est utilisée pour envisager la possibilité de supporter d'autres SGBD.

La classe « ManageSQLite » permet de gérer spécifiquement notre base de données SQLite.

En cas d'erreur lors de la création, une exception est déclenchée.

INTERFACES « DAO » ET CLASSES « SQLITEDAO »

Les opérations nécessaires au chargement et à l'enregistrement des données sont implémentées par le biais du design pattern « Data Access Object » au niveau des interfaces « IDaoContact », « IDaoInteraction », « IDaoTodo » et « IDaoGeneral ». Ces interfaces fournissent les opérations CRUD et sont spécifiques à un type de modèle en particulier de la couche métier.

L'utilisation du design pattern « DAO » permet de passer facilement du modèle objet au modèle relationnel et inversement, tout en permettant la séparation de la classe qui gère la persistance de la classe qui stocke les données (modèles) pour respecter le principe objet de « Responsabilité unique ». Par ailleurs, cela permet d'envisager l'utilisation de différents SGBD ou d'autres moyens de persistance comme JSON ou XML.

La méthode « readMaxId() » permet de retourner l'identifiant maximal jamais généré pour le type de modèle concerné (cela sélectionne l'identifiant maximal dans la table de la base de données), voir le rapport du jalon n°1, partie « Identifiant des modèles ».

Les classes « SQLiteDao » implémentent ces interfaces pour un SGBD de type SQLite spécifiquement.

En cas d'erreur lors d'une des opérations, une exception est déclenchée.

Cas de l'interface « IDaoGeneral »

Cette interface ne concerne aucun type de modèle, mais permet simplement d'enregistrer dans la base de données, un attribut quelconque, d'où l'utilisation d'un « QVariant » pour la valeur de l'attribut. Voir la partie Schéma relationnel de la base de données – Table « Général » pour plus d'informations.

CLASSE « ADUMP » ET CLASSE « DUMPJSON »

La classe abstraite « ADump » permet d'effectuer un export des données de l'application dans un fichier. Elle possède une méthode abstraite pour exporter toutes les données de l'application. L'utilisation d'une classe abstraite permet d'envisager le support de différents formats comme JSON, XML, etc.

La classe fille « DumpJson » permet d'exporter au format JSON. Les méthodes privées de cette classe permettent d'exporter un type d'instance en particulier. Enfin, cette classe utilise la classe Qt « QJsonDocument » pour effectuer la création du document JSON, et la classe Qt « QFile » pour écrire le fichier à partir du document JSON.

En cas d'erreur lors de l'export, une exception est déclenchée.

CLASSE « STDQT »

Cette classe statique est utilitaire pour faciliter la conversion de type C++ standard en type Qt et inversement. Elle permet principalement de convertir un string ou une date. À noter, qu'il y a deux méthodes surchargées utilisant un « QVariant » en paramètre, cela est utile pour facilement convertir les données reçues depuis les requêtes SQL de Qt puisqu'elles retournent des « QVariant ». Si un « QVariant » ne peut pas être converti, une exception est déclenchée.

COUCHE IHM

La couche IHM étant vaste, nous allons l'étudier plus en détail dans la prochaine partie. Cependant, nous pouvons déjà en faire un premier aperçu.

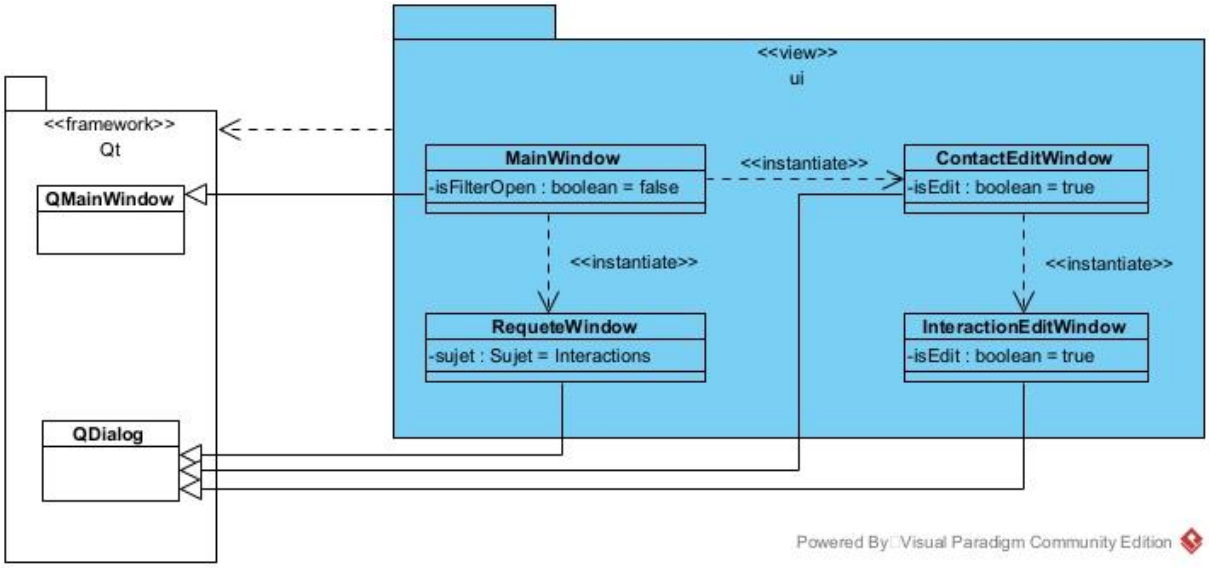


Diagramme 4 : Diagramme de classes d'aperçu de la couche IHM.

Comme vous pouvez le voir, nous avons au total, quatre fenêtres. La fenêtre principale hérite d'une « QMainWindow » pour bénéficier d'un menu et d'une barre de statuts. De l'autre côté, nous avons les trois autres fenêtres qui héritent d'une « QDialog » et sont modales.

La fenêtre principale « MainWindow » ouvre une fenêtre de requête « RequeteWindow » et une fenêtre d'édition d'un contact « ContactEditWindow ».

La fenêtre d'édition d'un contact « ContactEditWindow » ouvre une fenêtre d'édition d'une interaction « InterationEditWindow ».

DETAILS DE LA COUCHE IHM

Cette couche constitue l'interface utilisateur avec les différentes fenêtres. Nous avons au total, quatre fenêtres, que nous allons voir une par une.

COMMUNS A TOUTES LES FENETRES

Toutes les fenêtres ont un lien avec la classe « StdQt » de la couche de stockage pour effectuer des conversions entre type C++ standard et type Qt et inversement. Ceci pour lire et affecter les données des modèles.

Par ailleurs, la plupart des fenêtres ont un lien avec la classe « QMessageBox » du framework Qt pour afficher des dialogues d'information, de question, et d'erreur.

Les descriptifs de chacune des méthodes sont disponibles dans la documentation Doxygen de la couche IHM.

FENETRE PRINCIPALE

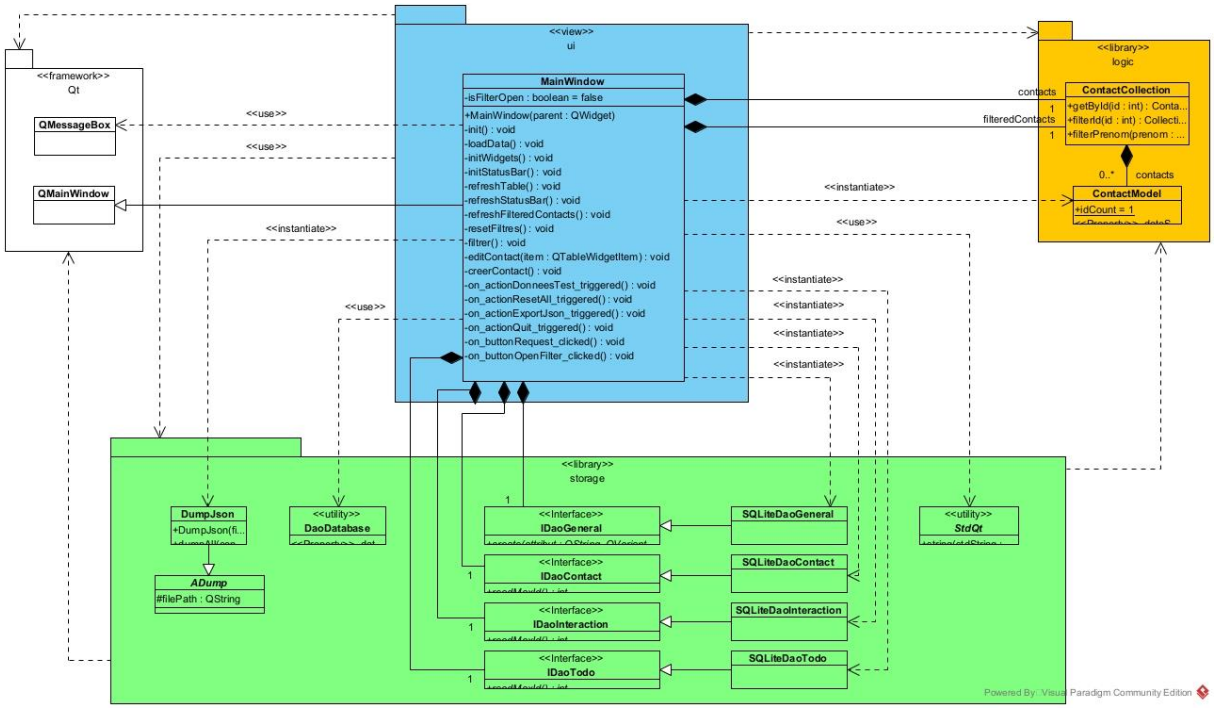


Diagramme 5 : Diagramme de classes de la fenêtre principale.

LIENS AVEC LA COUCHE METIER

La fenêtre contient une liste « contacts » contenant l'intégralité des contacts de l'application, tandis qu'une autre liste « filteredContacts » contient les contacts filtrés par la recherche de l'utilisateur.

Les contacts sont chargés au moment de l'ouverture de la fenêtre, dans la méthode « init() » et « loadData() ».

La fenêtre instancie un « ContactModel » lors de l'ouverture de la fenêtre de création d'un contact, l'instance lui est passée en référence pour que cette dernière remplisse le contact.

LIENS AVEC LE COUCHE DE STOCKAGE

La fenêtre contient tous les types d'instance de classe « DAO » pour créer, modifier et supprimer les contacts quand l'utilisateur effectue des actions. Cela permet également de charger les données au démarrage de l'application, c'est-à-dire, contacts, mais aussi interactions et todos, d'où la nécessité de tous les « DAO ».

La fenêtre initialise la connexion à la base de données au démarrage via la classe « DaoDatabase » et permet d'exporter en JSON depuis le menu en utilisant la classe « DumpJson ».

FENETRE DE REQUETE

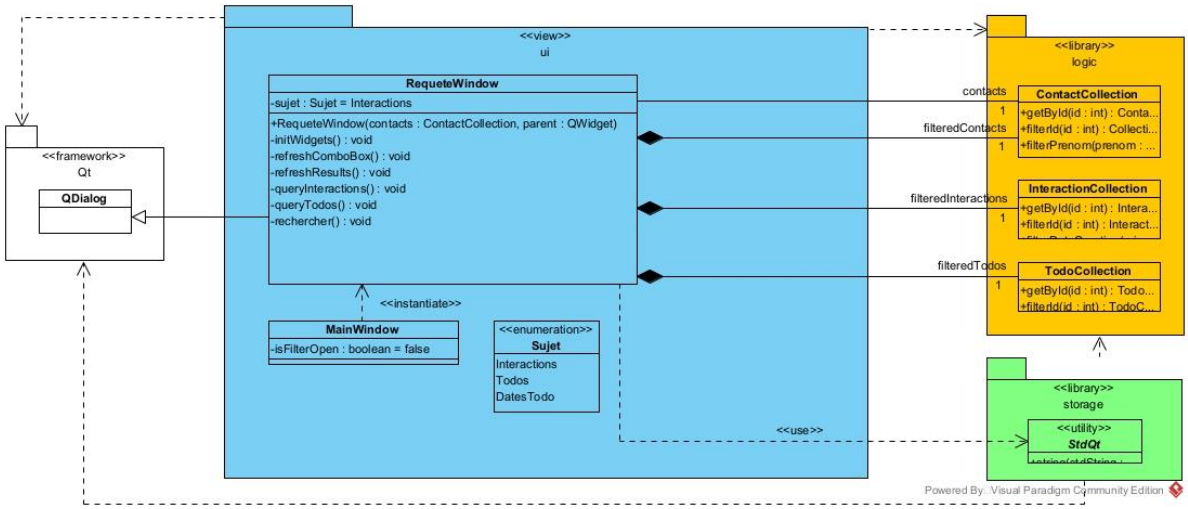


Diagramme 6 : Diagrammes de classe de la fenêtre de requête.

La fenêtre principale instancie et ouvre cette fenêtre en lui passant la liste de la totalité des contacts.

LIENS AVEC LA COUCHE METIER

La liste « contacts » est donnée par la fenêtre principale au moment de l'instanciation. La liste « contacts » est affichée dans une liste déroulante.

La fenêtre contient également la liste « filteredContacts » des contacts filtrés, la liste « filteredInteractions » des interactions filtrées, et la liste « filteredTodos » des todos filtrés. Ces trois dernières listes filtrées sont issues de la requête demandée par l'utilisateur.

SUJET DE RECHERCHE ET REQUETE

L'énumération « Sujet » permet de préciser le sujet de la requête : si l'on souhaite rechercher des interactions, des todos ou bien des dates de todo. Un seul type de sujet est possible par requête.

Ainsi, en fonction du sujet choisi, soit la méthode « queryInteractions() », soit la méthode « queryTodos() » est appelée pour exécuter la requête.

Les requêtes s'effectuent en utilisant les méthodes de filtres des collections.

FENETRE D'EDITION D'UN CONTACT

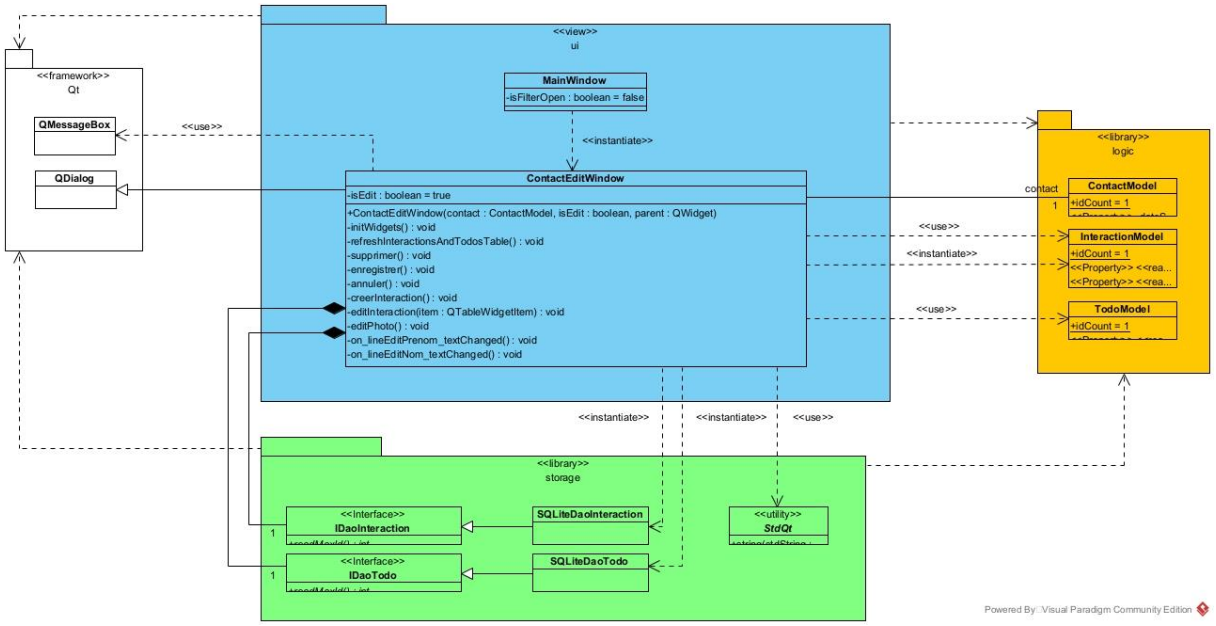


Diagramme 7 : Diagramme de classes de la fenêtre d'édition d'un contact.

La fenêtre principale instancie et ouvre cette fenêtre en lui passant le contact à remplir ou à modifier si la fenêtre est instanciée en mode « création » ou en mode « édition » (via le paramètre du constructeur « isEdit »).

LIENS AVEC LA COUCHE METIER

La fenêtre contient le « contact » à remplir ou à modifier fournit par la fenêtre principale. Cela lui permet également d'afficher les données du contact si celui-ci existait déjà. La fenêtre utilise également les classes « InteractionModel » et « TodoModel » pour afficher les interactions et les todos du contact.

La fenêtre instancie un « InteractionModel » lors de l'ouverture de la fenêtre de création d'une interaction, l'instance lui est passée en référence pour que cette dernière remplisse l'interaction.

LIENS AVEC COUCHE DE STOCKAGE

La fenêtre créant et modifiant des interactions et des todos, elle doit les enregistrer dans la base de données en passant par les « DAO ».

PHOTO DU CONTACT

Le chemin de la photo du contact est récupéré depuis l'instance « contact ». Si le chemin est vide, alors une photo par défaut est affichée. **La photo par défaut a été intégrée dans un fichier ressource Qt**, pour éviter de « balader » un fichier image externe à l'exécutable de l'application.

Lorsque le fichier d'une photo est chargé, celui-ci est copié dans un répertoire « photos » à l'endroit de l'exécutable de l'application. Le chemin de la photo est ensuite affecté au contact.

FENETRE D'EDITION D'UNE INTERACTION

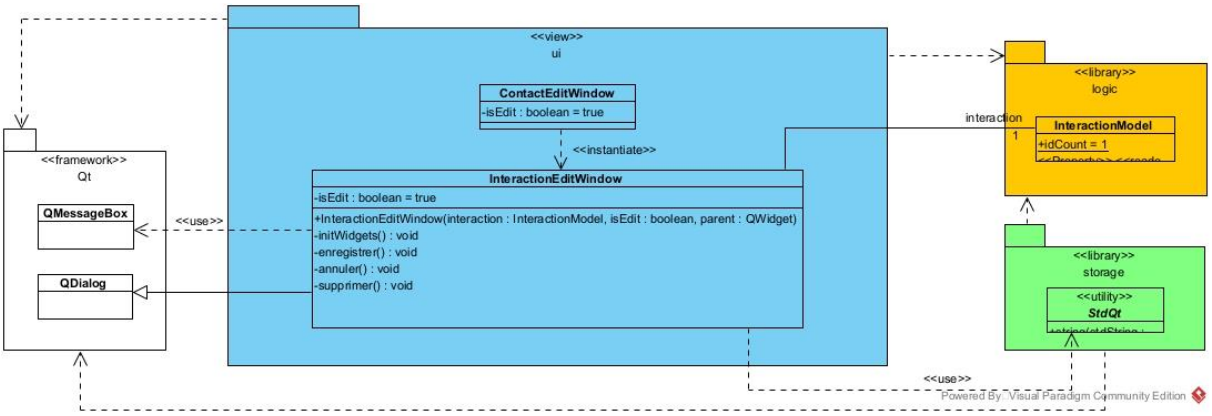


Diagramme 8 : Diagramme de classes de la fenêtre d'édition d'une interaction.

La fenêtre d'édition d'un contact instancie et ouvre cette fenêtre en lui passant l'interaction à remplir ou à modifier si la fenêtre est instanciée en mode « création » ou en mode « édition » (via le paramètre du constructeur « isEdit »).

LIENS AVEC LA COUCHE METIER

La fenêtre contient « interaction » à remplir ou à modifier fournit par la fenêtre d'édition d'un contact. Cela lui permet également d'afficher les données de l'interaction si celle-ci existait déjà. Les todos sont automatiquement instanciés en appelant la méthode « parseTodos() » de la classe « InteractionModel ».

DIAGRAMMES DE SEQUENCES

Dans cette partie, nous allons voir trois diagrammes de séquence modélisant l'implémentation de certains mécanismes que nous trouvons intéressant de montrer.

INITIALISATION DE L'APPLICATION

Ce diagramme de séquence montre l'initialisation de l'application au démarrage et la création de la fenêtre principale. Il montre principalement la connexion à la base de données et le chargement des données depuis celle-ci.

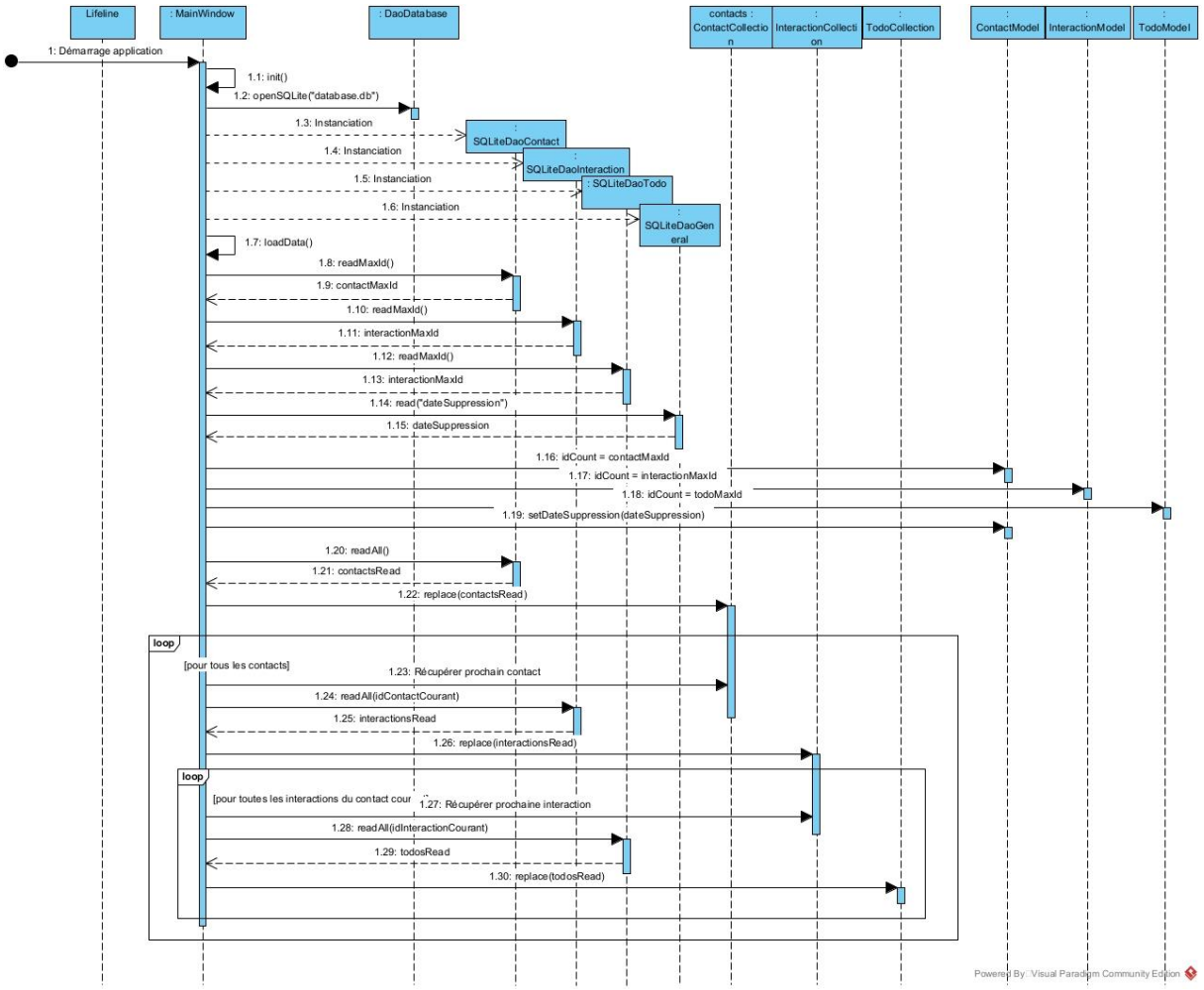


Diagramme 9 : Diagramme de séquence de l'initialisation de l'application.

RECHERCHE DE CONTACTS

Ce diagramme de séquence montre le mécanisme de recherche de contacts selon certains critères dans la fenêtre principale.

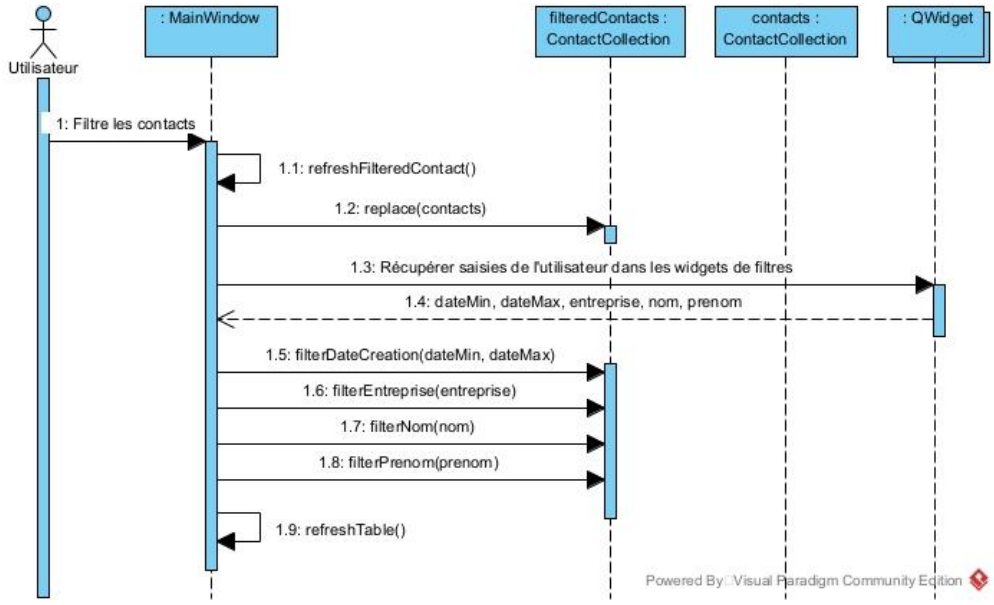


Diagramme 10 : Diagramme de séquence de la recherche de contacts.

ÉDITION D'UN CONTACT

Ce diagramme de séquence montre le mécanisme lorsque l'utilisateur double clique sur un contact dans la liste dans la fenêtre principale. Cette action ouvre la fenêtre d'édition du contact, et effectue un processus selon le bouton cliqué précédent la fermeture de la fenêtre (« Enregistrer » ou « Supprimer »). Si le bouton cliqué est « Annuler », l'instance du contact n'est pas modifiée.

Par ailleurs, l'instance « item » est la « ligne » double cliquée dans la liste de contacts.

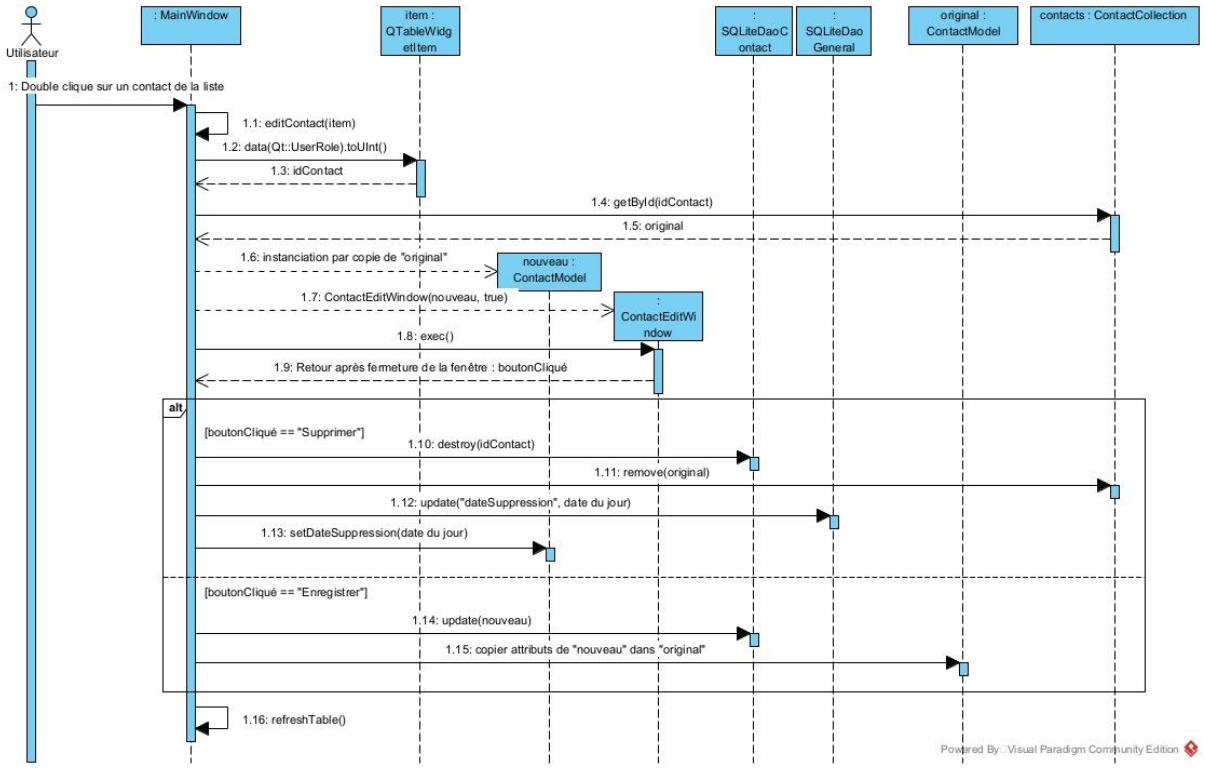


Diagramme 11 : Diagramme de séquence de l'édition d'un contact à partir de la liste de la fenêtre principale.

Nous avons procédé à peu près de la même façon pour la création d'un contact, d'une interaction ou la modification d'une interaction.

Note concernant le message « 1.2 »

Nous avons stocké l'identifiant du contact dans l'item (la ligne) de la table qui lui correspond pour qu'on puisse le récupérer plus tard comme ici. L'identifiant a donc été stocké dans les données de type « UserRole » de l'item. Ce type de données est d'après la documentation de Qt utilisé pour stocker ses propres données applicatives.

SCHEMA RELATIONNEL DE LA BASE DE DONNEES

La structure de la base de données est une simple transcription des classes modèles. Les relations sont en forme normale de Boyce-Codd.

SCHEMA RELATIONNEL

Contact(id : int, nom : text, prenom : text, email : text, tel : text, photo : text, date_modification : text, date_creation : text)

Interaction(id : int, contenu : text, date_creation : text, id_contact : int)

Todo(id : int, resume : text, date_realisation : text, id_interaction : int)

General(attribut : text, valeur : text)

CONTRAINTES

La base de données possède des contraintes sur certains attributs, par exemple « Contact.nom », « Contact.prenom », « Contact.date_creation », « Interaction.contenu », et « Todo.resume » ne peuvent pas être nuls ou vides.

La date du jour est automatiquement affectée par défaut aux dates de création et à la date de réalisation.

CLES ETRANGERES

Les clés étrangères ont été configurées pour supprimer automatiquement tout enfant d'un parent supprimé. Cela permet de faciliter les suppressions au niveau de l'application : il suffit de simplement supprimer tous les contacts pour supprimer automatiquement les interactions et les todos.

TABLE « GENERAL »

Cette table permet de stocker la valeur d'un attribut quelconque. Elle s'apparente à une carte clé-valeur. L'utilité est de stocker la date de dernière suppression qui est un attribut global à l'application et ne dépend d'aucune instance.

SIGNAUX, SLOTS ET EVENEMENTS

La majorité de l'interface utilisateur a été réalisée avec le Qt Designer étant donné que le sujet ne précisait aucune interdiction sur cet outil. Nous avons donc utilisé l'outil d'association de signal-slot du Qt Designer pour des actions simples comme « cocher une case → activer/désactiver le champ de texte ».

Autrement, pour des actions plus complexes, nous avons créé nos propres slots et les avons connectés à des widgets comme des boutons. Nous n'avons pas jugé nécessaire, ni aurait eu l'utilité de créer nos propres signaux.

FENETRE PRINCIPALE

Émetteur	Signal	Destinataire	Slot	Paramètre(s)
Bouton « Réinitialiser filtres »	clicked	this	resetFiltres	
Bouton « Filtrer »	clicked	this	filtrer	
Table de la liste des contacts	doubleClicked	this	editContact	QTableWidgetItem : item double cliqué, contient l'identifiant du contact.
Bouton « Créer contact »	clicked	this	creerContact	
Menu action « Insérer données test »	triggered	this	actionDonneesTest	
Menu action « Réinitialiser tout »	triggered	this	actionResetAll	
Menu action « Exporter en JSON »	triggered	this	actionExportJson	
Menu action « Quitter »	triggered	this	actionQuit	
Bouton « Requête »	clicked	this	buttonRequest	
Bouton « Ouvrir filtres »	clicked	this	buttonOpenFilter	

Tableau 1 : Signaux et slots de la fenêtre principale.

FENETRE DE REQUETE

Émetteur	Signal	Destinataire	Slot	Paramètre(s)
Bouton « Rechercher »	clicked	this	rechercher	
Radio « Interactions »	toggled	this	radioInteractions	bool : si la case est cochée, on modifie le sujet de recherche.
Radio « Todos »	toggled	this	radioTodos	bool : si la case est cochée, on modifie le sujet de recherche.
Radio « Dates todo »	toggled	this	radioDatesTodo	bool : si la case est cochée, on modifie le sujet de recherche.

Tableau 2 : Signaux et slots de la fenêtre de requête.

FENETRE D'EDITION D'UN CONTACT

Émetteur	Signal	Destinataire	Slot	Paramètre(s)
Bouton « Supprimer »	clicked	this	supprimer	
Bouton « Enregistrer »	clicked	this	enregistrer	
Bouton « Annuler »	clicked	this	annuler	
Bouton « Créer interaction »	clicked	this	creerInteraction	
Table de la liste d'interactions	doubleClicked	this	editInteraction	QTableWidgetItem : item double cliqué, contient l'identifiant de l'interaction.
Bouton de la photo	clicked	this	editPhoto	
Champ de texte « Prénom »	textChanged	this	lineEditPrenom	
Champ de texte « Nom »	textChanged	this	lineEditNom	

Tableau 3 : Signaux et slots de la fenêtre d'édition d'un contact.

FENETRE D'EDITION D'UNE INTERACTION

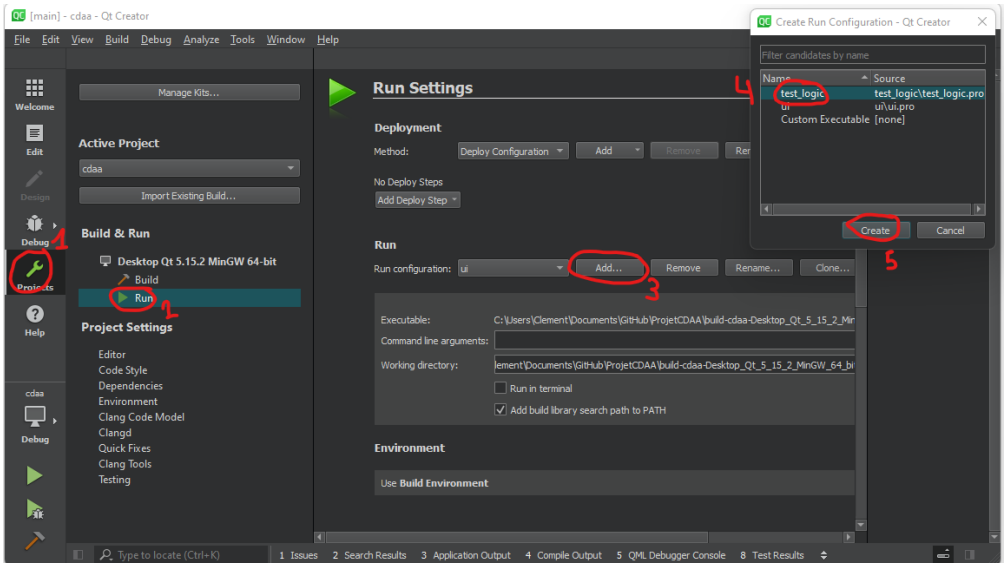
Émetteur	Signal	Destinataire	Slot	Paramètre(s)
Bouton « Enregistrer »	clicked	this	enregistrer	
Bouton « Annuler »	clicked	this	annuler	
Bouton « Supprimer »	clicked	this	supprimer	

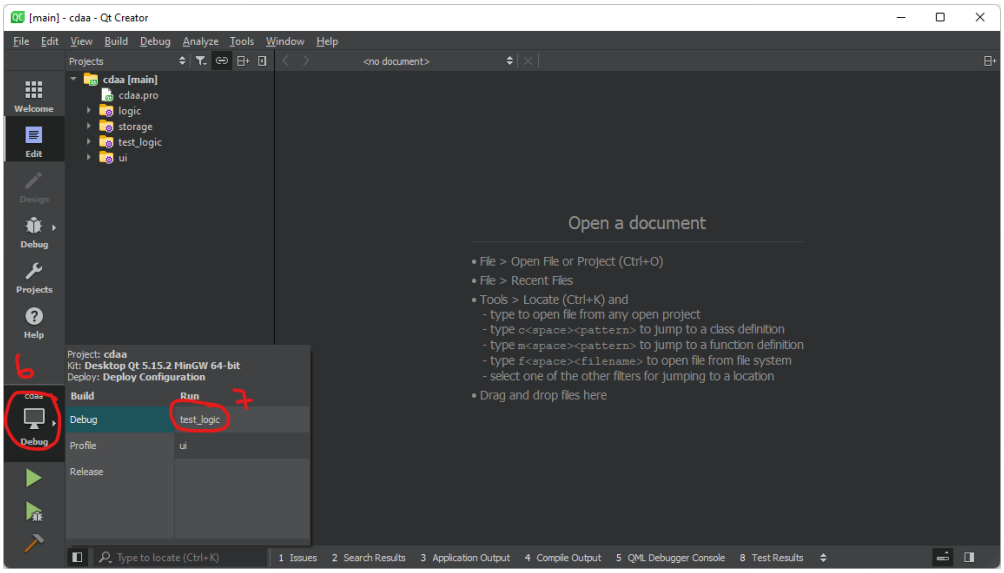
Tableau 4 : Signaux et slots de la fenêtre d'édition d'une interaction.

INSTRUCTIONS DE COMPILATION

Les fichiers sources se trouvent dans le répertoire « cdaa ». Pour compiler le projet, il suffit d'ouvrir le fichier « cdaa/cdaa.pro » dans Qt Creator, et de build le projet.

Si vous souhaitez exécuter les tests unitaires de la couche métier, il faut se rendre dans « Projects » puis « Run » et d'ajouter la configuration d'exécution pour le package « test_logic » :





La documentation Doxygen a été entièrement écrite pour chacun des couches. Vous pouvez générer vous-même les documentations au format HTML en exécutant le script « run_doxygen.sh » sous Linux ou « run_doxygen.bat » sous Windows dans le répertoire « docs ».

INSTRUCTIONS D'UTILISATION

L'application est utilisable sous Windows (le projet a été développé avec ce système d'exploitation, comme en témoignent les captures d'écran dans la partie Fonctionnalités implémentées) et sous Linux.

La base de données n'a pas besoin d'être initialisée manuellement, tout est géré par l'application. Il vous suffit simplement de la démarrer et de l'utiliser. Notez que vous pouvez insérer des données de test dans le menu « Fichier » de la fenêtre principale.

TABLES DES IMAGES

TABLES DES DIAGRAMMES

DIAGRAMMES DES TABLEAUX