Conception et développement avancé d’applications

RAPPORT DU PROJET

Mohamed BOUCHENGUOUR
Mehdi ASNI
TP GROUPE 6 | GROUPE AB

ANNEE UNIVERSITAIRE 2022/2023

Table des matières

1. Conception de l'application

Diagramme de classe :

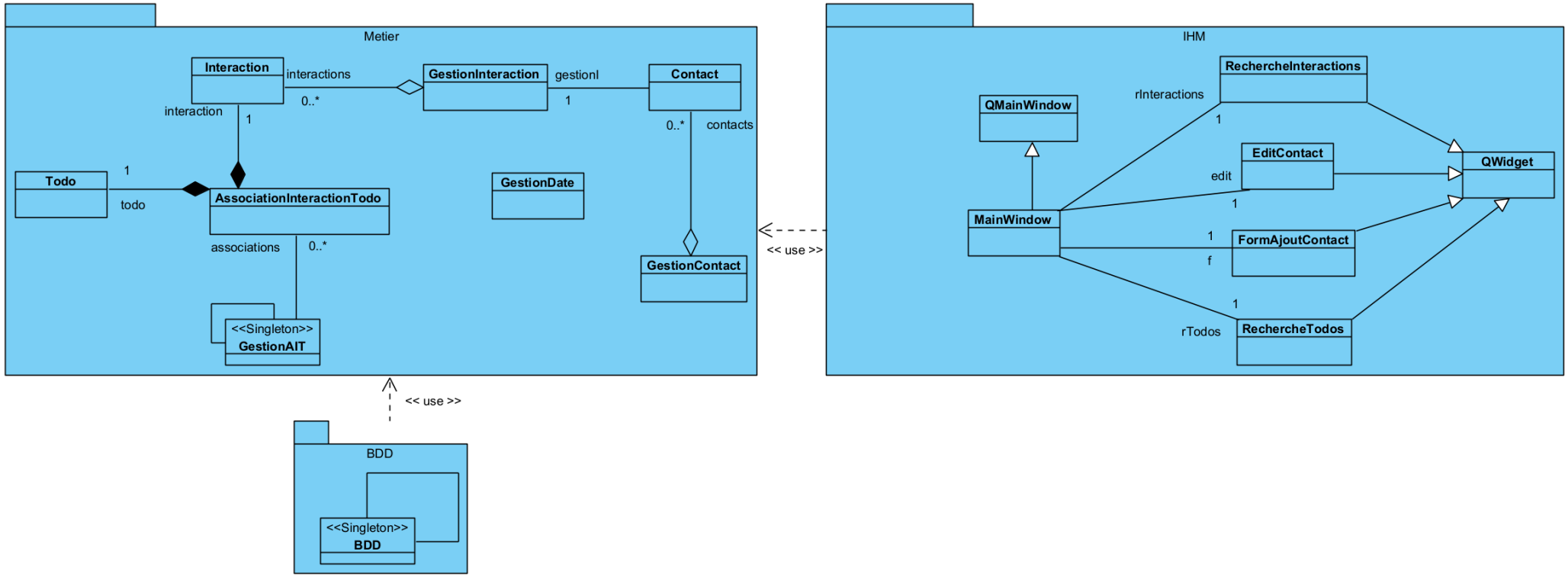

Figure 1 - Diagramme de classe de notre application

2. Descriptions et justificatifs des classes

1. Justificatif global

- Les classes **Contact**, **Interaction**, **Todo** et **AssociationInteractionTodo** ont un id qui nous permettra de les associer à la base de données (par exemple : le contact ayant l'id 5 correspondra au contact ayant l'id 5 dans la base de données). Accéder à une instance dans une liste et la modifier sera plus simple pour l'application et pour mettre à jour la base de données. Dans les classes, ces id ont un getter mais également un setter afin de modifier l'id de l'instance avec l'id de la base de données après une insertion.
- Pour manipuler les dates, nous avons décidé de créer une classe qui nous permettra de les gérer (GestionDate).
- Pour manipuler la base de données, nous avons décidé de créer un singleton ce qui nous permettra de manipuler la base de données dans n'importe quel widget.
- Un todo et une interaction seront liés par le biais de la classe AssociationInteractionTodo. Cette classe stockera l'adresses un todo et l'adresses de l'interaction correspondant.
- La liste de toutes les AssociationInteractionTodo sera stockée dans un singleton afin d'avoir un « annuaire ». On retrouvera facilement les todo depuis une interaction et une interaction depuis un todo. L'initialisation des AssociationInteractionTodo avec la base de données et la manipulation globale des Todo/Interaction seront également simplifiés. Todo, Interaction et contact sont ainsi indépendants.

2. Classe GestionDate

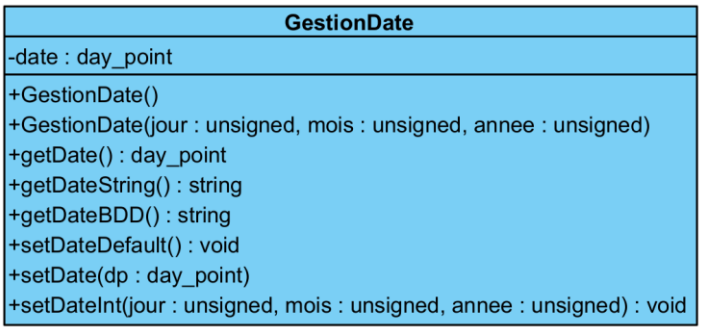

Figure 2- Classe GestionDate

Description

Pour manipuler les dates, nous avons décidé de créer une classe à partir du fichier date.h fournis qui nous permet de stocker et de manipuler des dates. Cependant, la modification d'une date nécessite plusieurs lignes et n'est pas forcément intuitif. Pour pallier ce problème, la classe Date nous permettra de gérer une date avec des fonctions plus simples. Par exemple, la fonction setDateInt(unisgned jour, unsigned mois,unsigned annee) nous permettra d'avoir une date avec le jour, le mois et l'année voulus.

Attributs

- **date :** type day_point, représente la date de la classe

Méthodes

- **+GestionDate**() : constructeur par défaut, initialise l'attribut date avec la date du jour
- **+GestionDate**(jour : unsigned, mois : unsigned, annee : unsigned) : constructeur avec le jour, le mois et l'année voulus en paramètres
- **+getDate :** getter de l'attribut date.
- **+getDateString :** permet de récupérer l'attribut date, en type std::string et en format 'JJ/MM/AAAA'.
- **+getDateBDD :** permet de récupérer l'attribut date, en type std::string et en format 'AAAA-MM-JJ' afin d'insérer une date dans la base de données.
- **+setDateDefaut :** affecte la date du jour à l'attribut date.
- **+setDate :** setter de l'attribut dp.
- **+setDateInt :** setter de l'attribut dp avec le jour, le mois et l'année en paramètre.

3. Classe Contact

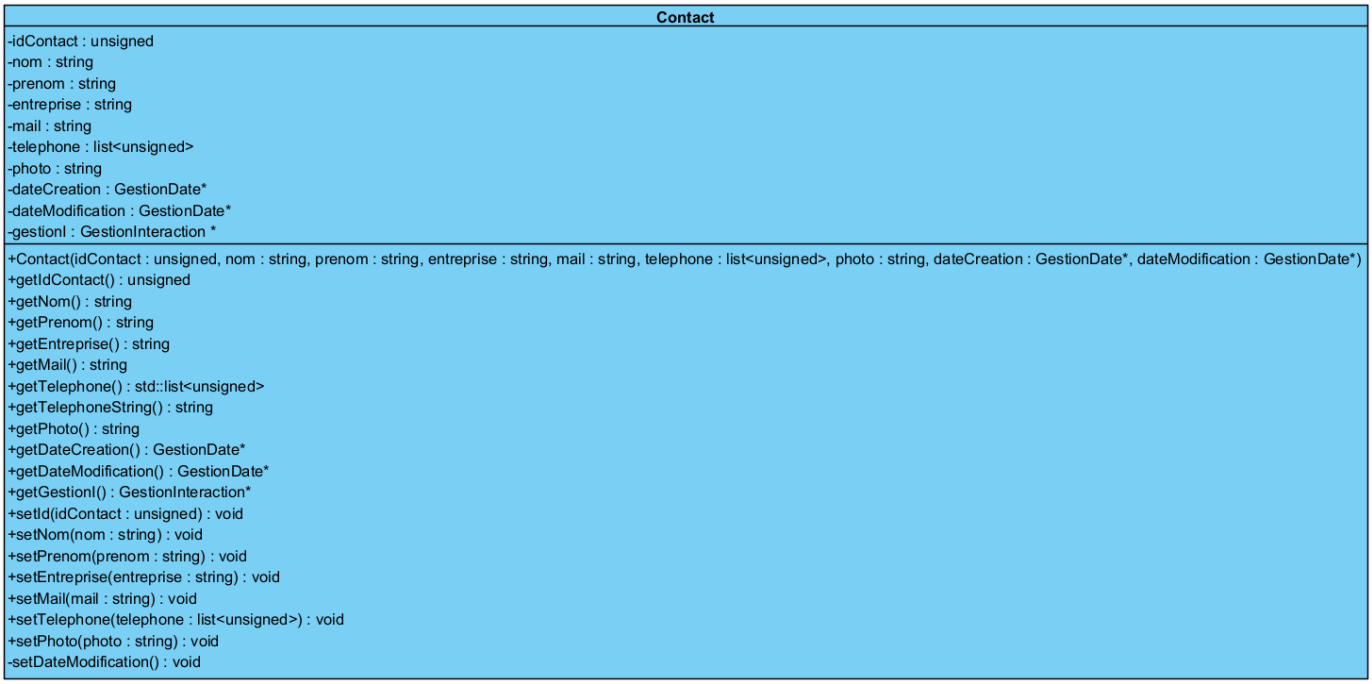

Figure 3 - Classe Contact

Description

Dans notre application, la classe correspondant à un contact ne gère que ses propres informations. Dans cette classe, nous stockons les différentes informations que nous pouvons récupérer et modifier à l'aide de getter et setter. Le téléphone est stocké dans une liste d'unsigned pour éviter les mauvaises insertions comme des caractères par exemple. Nous avons deux dates, la date de modification (attribut dateModification) et la date de création du contact (attribut dateCreation *). Ces dates seront de type GestionDate* afin de pouvoir les gérer facilement (notamment en cas de modification pour la date de modification). Contact peut être initialisé sans passer de date au constructeur (en mettant nullptr). L'attribut dateModification sera gardé à nullptr et l'attribut dateCreation sera initialisé avec la date du jour. Les setters, en plus de modifier l'attribut correspondant, modifient la date de modification avec la date du jour. Cependant, dateModification et dateCreation n'ont pas de setter car la date de création est fixe et la date de modification ne peut être modifiée que si un attribut est modifié. La classe contact contient également un attribut de type GestionInteraction* (gestionI), classe qui permet de stocker et de manipuler les interactions du contact.

Attributs

- **idContact :** type unsigned, représente l'identifiant du contact.
- **nom :** type string, représente le nom du contact.
- **prenom :** type string, représente le prénom du contact.
- **entreprise :** type int, représente le nom de l'entreprise.
- **mail :** type string, représente le mail du contact.
- **telephone :** type string, représente le numéro de téléphone du contact.
- **photo :** type string, représente l'URI de la photo du contact.
- **dateCreation :** type GestionDate*, représente la date de création du contact.
- **dateModification :** type GestionDate*, représente la date de la dernière modification du contact.

- **gestionI :** type GestionInteraction*, instance qui permet de stocker et manipuler des interactions.

Méthodes

- **+Contact(idContact : unsigned, nom : string, prenom : string, entreprise : string, mail : string, telephone : list<unsigned>, photo : string, dateCreation : GestionDate*, dateModification : GestionDate*) :** constructeur de la classe Contact avec en paramètres toutes les informations du contact. Initialise dateCreation à la date du jour si le paramètre dateCreation du constructeur est nullptr.
- **+getIdContact :** getter de l'attribut idContact.
- **+getNom :** getter de l'attribut nom.
- **+getPrenom :** getter de l'attribut prenom.
- **+getEntreprise :** getter de l'attribut entreprise.
- **+getMail :** setter de l'attribut mail.
- **+getTelephone :** getter de l'attribut telephone.
- **+getTelephoneString :** getter de l'attribut telephone en type std ::string.
- **+getPhoto :** getter de l'attribut photo.
- **+getDateCreation :** getter de l'attribut dateCreation.
- **+getDateModification :** getter de l'attribut dateModification.
- **+getGestionI :** getter de l'instance GestionAIT.

- **+setIdContact :** setter de l'attribut idContact.
- **+setNom :** setter de l'attribut nom. A chaque appel, modifie la date de dateModification avec la date du jour.
- **+setPrenom :** setter de l'attribut prenom. A chaque appel, modifie la date de dateModification avec la date du jour.
- **+setEntreprise :** setter de l'attribut entreprise. A chaque appel, modifie la date de dateModification avec la date du jour.
- **+setMail :** setter de l'attribut mail. A chaque appel, modifie la date de dateModification avec la date du jour.
- **+setTelephone :** setter de l'attribut telephone. A chaque appel, modifie la date de dateModification avec la date du jour.
- **+setPhoto :** setter de l'attribut photo. A chaque appel, modifie la date de dateModification avec la date du jour.

4. Classe Interaction

Interaction
-idInteraction : unsigned -contenuInteraction : string -dateAjout : GestionDate*
+Interaction(idInteraction : unsigned, contenuInteraction : string, dateAjout : GestionDate*) +getIdInteraction() : unsigned +getContenuInteraction() : string +getDateAjout() : GestionDate* +setIdInteraction(id : unsigned) : void +setContenuInteraction(contenuInteraction : string) : void +setDateAjout(dateAjout : GestionDate*) : void

Figure 4 - Classe Interaction

Description

Cette classe stock l'id, le contenu et la date de création d'une interaction. Une interaction sera stockée dans une liste d'Interaction, elle-même stockée dans une instance GestionInteraction d'un contact.

Attributs

- **idInteraction :** type unsigned, représente l'identifiant de l'interaction.
- **contenuInteraction :** type string, représente le contenu de l'interaction.
- **dateAjout :** type GestionDate*, représente la date d'ajout de l'interaction.

Méthodes

- **+Interaction(idInteraction : unsigned, contenuInteraction : string, dateAjout : GestionDate*) :** constructeur avec l'id, le contenu et la date d'ajout de l'Interaction en paramètre.
- **+getIdInteraction :** getter de l'attribut idInteraction.
- **+getContenuInteraction :** getter de l'attribut contenuInteraction.
- **+getDateAjout :** getter de l'attribut dateAjout.
- **+setIdInteraction :** setter de l'attribut idInteraction. Utiliser lors de l'insertion
- **+setContenuInteraction :** setter de l'attribut contenuInteraction.
- **+setDateAjout :** setter de l'attribut dateAjout.

5. Classe Todo

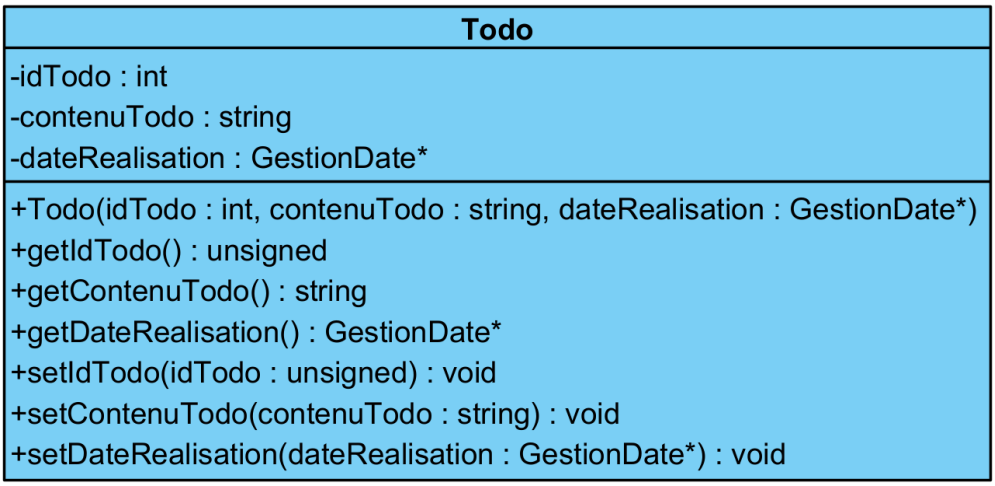

Figure 5 - Classe Todo

Description

Cette classe stock le contenu et la date de réalisation d'un todo. La date de réalisation est initialisée à la date du jour si elle n'est pas précisée.

Attributs

- **idTodo :** type int, représente l'identifiant du Todo
- **contenuTodo :** type string, représente le contenu du Todo
- **dateRealisation :** type GestionDate*, représente la date de réalisation du todo

Méthodes

- **+Todo(idTodo : int, contenuTodo : string, dateRealisation : GestionDate*):** constructeur avec l'id, le contenu et la date d'ajout du Todo en paramètre. Initialise dateRealisation à la date du jour si le paramètre dateRealisaion du constructeur est nullptr.
- **+getIdTodo :** getter de l'attribut idTodo**.**
- **+getcontenuTodo :** getter de l'attribut contenuTodo**.**
- **+getDateRealisation :** getter de l'attribut dateRealisation**.**
- **+setIdTodo :** setter de l'attribut idTodo**.**
- **+setContenuTodo :** setter de l'attribut contenuTodo.
- **+setDateRealisation :** setter de l'attribut dateRealisation.

6. Classe AssociationInteractionTodo

AssociationInteractionTodo
-idAssociation : unsigned -todo : Todo* -interaction : Interaction*
+AssociationInteractionTodo(idAssociation : unsigned, todo : Todo*, interaction : Interaction*) +setIdAssociation(idAssociation : unsigned) : void +getIdAssociation() : unsigned +getTodo() : Todo* +getInteraction() : Interaction*

Figure 6 - Classe AssociationInteractionTodo

Description

Cette classe stock l'adresse d'un todo et d'une interaction. Cette classe nous permettra donc de faire l'association entre les deux afin d'avoir l'interaction correspondant à un todo et inversement.

Attributs

- **idAssociation :** type unsigned, représente l'identifiant de l'association
- **todo :** adresse d'un Todo
- **interaction :** adresse d'une Interaction

Méthodes

- **+AssociationInteractionTodo(idAssociation : unsigned, todo : Todo*, interaction : Interaction*) :** constructeur avec l'id d'AssociationInteractionTodo, l'adresse du Todo et l'adresse de l'Interaction
- **+setIdAssociation :** setter de l'attribut idAssociation.
- **+getTodo :** getter de l'attribut todo.
- **+getInteraction :** getter de l'attribut interaction.

Dans cette application, nous avons besoin de stocker plusieurs listes : une liste de Contact, d'Interaction et d'AssociationInteractionTodo. Pour pouvoir les gérer efficacement et ne pas avoir de forte dépendance, nous avons décidé de créer une classe pour gérer chaque liste : GestionInteraction pour la liste d'Interaction, GestionContact pour la liste de Contact et GestionAit pour la liste d'AssociationInteractionTodo.

7. Classe GestionContact

GestionContact
-dateSuppression : GestionDate*
+GestionContact(date : GestionDate*) +setContacts(contacts : std::list<Contact*>) : void +getContacts() : list<Contacts> +getDateSuppression() : GestionDate* +getNbContact() : int +addContact(contact : Contact*) : void +getById(idContact : unsigned) : Contact* +getAllByName(nom : string) : std::list<Contact*> +getAllByEntrepise(entreprise : string) : std::list<Contact*> +getByDate(dateDebut : GestionDate*, dateFin : GestionDate*) : std::list<Contact*> +getByDatesIntervalle(dateD : GestionDate*, DateF : GestionDate*) : std::list<Contact*> +removeByIdContact(idContact : unsigned) : void

Figure 7 - Classe GestionContact

Description

Cette classe nous permettra de gérer la liste des contacts en stockant leurs adresses dans un std::list. Elle stocke également la date de la dernière suppression d'un contact (dateSuppression de type GestionDate*). Cette date a un getter mais pas de setter car la date sera modifiée à la date du jour uniquement lors de la suppression d'un contact. Cette classe nous permet d'ajouter un contact à la liste, de chercher un contact à partir de son id, d'un nom, d'une entreprise. Récupérer tous les contacts d'une entreprise et également possible.

Lors de la suppression d'un contact (possible à partir d'un id, d'un nom ou d'une entreprise), la date de suppression est mise à jour avec la date du jour. Ses interactions et Todo seront également supprimés de la liste d'association de gestionAIT.

Attributs

- **contacts** : std::list contenant les adresses des instances Contact.
- **dateSuppression :** type gestionDate*, représente la date de la dernière suppression d'un contact.

Méthodes

- **+GestionContact(GestionDate * const) :** constructeur avec la date de suppression passé en paramètre. On passe nullptr s'il n'y a pas de date de suppression.
- **+getContacts :** getter de l'attribut contacts.

- **+setContacts :** setter de l'attribut contacts.
- **+getDateSuppression :** getter de l'attribut dateSuppression.
- **+getNbContacts :** permet d'avoir le nombre de contacts.
- **+addContact :** ajoute un contact à la liste contacts.
- **+getById :** cherche et retourne un contact dans la liste contacts à l'aide de son id passé en paramètre.
- **+getAllByName :** retourne une liste contenant toutes les adresses des instances Contact ayant le nom passé en paramètre.
- **+getAllByEntreprise :** retourne une liste contenant toutes les adresses des instances Contact appartenant à l'entreprise passée en paramètre.
- **+getByDate** : retourne une liste contenant toutes les adresses des instances Contact créées à une date passé en paramètre.
- **+getByDatesIntervalle** : retourne une liste contenant toutes les adresses des instances Contact créées dans un intervalle de date passé en paramètre.
- **+removeByIdContact:** cherche et supprime un contact ayant l'idContact passé en paramètre.

8. Classe GestionInteraction

GestionInteraction
+GestionInteraction() +getInteractions() : std::list<Interaction> +setInteractions(interactions : std::list<Interaction>) +getInteractionById(idInteraction : unsigned) : Interaction* +addInteraction(interaction : Interaction) : void +removeInteractionById(idInteraction : unsigned) : void +getInteractionsByDate(date : GestionDate*) : std::list<Interaction> +getInteractionsByIntervalle(dateD : GestionDate*, dateF : GestionDate*) : std::list<Interaction> +getTodos() : list<Todo*>

Figure 8 - Classe GestionInteraction

Description

Cette classe nous permettra de gérer la liste des interactions d'un contact en les stockant dans un std::list. Cette classe nous permet également d'avoir tous les Todo correspondant aux interactions en les récupérant depuis la classe GestionAIT à l'aide la fonction getTodosByInteractions.

Attributs

- **std::list<Interaction> interactions:** std::list contenant des Interaction.

Méthodes

- **+GestionInteraction() :** constructeur par défaut.
- **+getInteractions :** getter de l'attribut interactions.
- **+setInteractions :** setter de l'attribut interactions.
- **+getInteractionById:** cherche et retourne l'adresse d'une Interaction à l'aide de son id passé en paramètre.
- **+addInteraction:** ajoute l'interaction passée en paramètre à la liste interactions.
- **+removeInteractionById() :** cherche et supprime une Interaction ayant l'idInteraction passé en paramètre.
- **+getInteractionsByDate:** retourne une liste contenant toutes les instances Interaction créées à une date passé en paramètre.
- **+getInteractionsByIntervalle:** retourne une liste contenant toutes les instances Interaction créées dans un intervalle de date passé en paramètre.

9. Classe GestionAIT

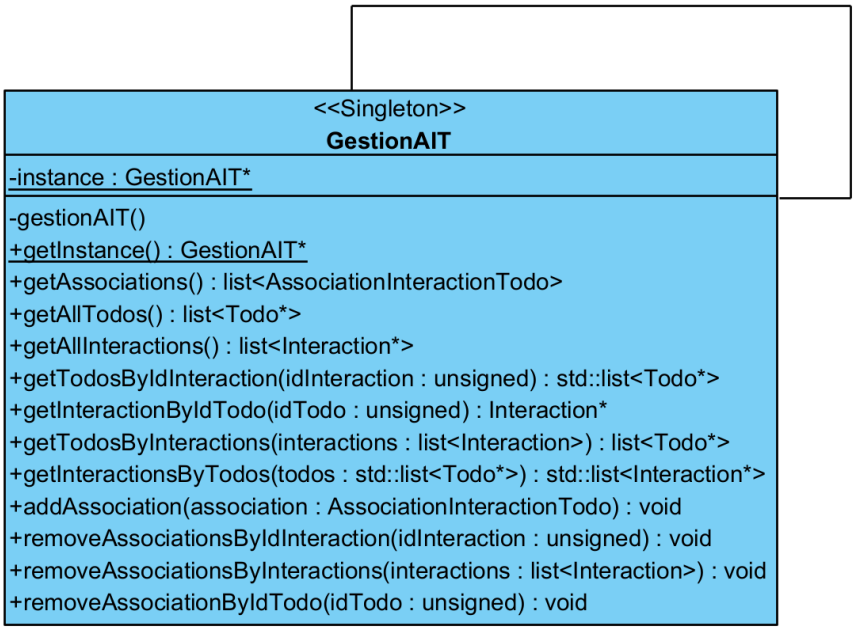

Figure 9 - Classe GestionAIT

Description

Pour pouvoir gérer les Todo et Interaction de manières globales, nous avons décidé de stocker toutes les associations dans la classe GestionAIT qui est un singleton. Nous n'avons pas créé une classe GestionAIT liée à chaque contact car cela crée une boucle dans notre UML. GestionAIT est donc accessibles par toutes les classes de l'application. Ce singleton fonctionne comme un annuaire, nous pourrons ainsi avoir accès à toutes les AssociationInteractionTodo, Todo et Interaction. Initialiser cette liste depuis la base de données se fera facilement mais également pour récupérer les Todo depuis une Interaction, une Interaction depuis un Todo mais également les Todo depuis une liste d'interactions (qui nous permettra d'afficher les Todo d'un contact). Ces fonctions envoient une liste d'adresses (adresse Todo, adresse Interaction, adresse AssociationInteractionTodo) afin de modifier l'instance directement et non une copie.

Attributs

- **instance :** type GestionAit*, singleton (static)
- **std::list<AssociationInteractionTodo> associations** : liste contenant les AssociationInteractionTodo

Méthodes

- **-gestionAIT :** constructeur par défaut, initialise l'attribut instance. Méthode privée.
- **+getInstance :** getter de l'attribut instance, appel le constructeur si l'attribut instance n'est pas initialisé.
- **+getAssociations:** getter de l'instance associations.
- **+getAllTodos :** récupère et renvoie dans une liste toutes les adresses des Todo contenus dans les AssociationInteractionTodo de l'attribut associations.
- **+ getAllInteractions:** récupère et renvoie dans une liste toutes les adresses des Interaction contenues dans les AssociationInteractionTodo de l'attribut associations.
- **+getTodosByIdInteraction:** récupère et renvoie toutes les adresses des Todo correspondant à une Interaction à l'aide de l'id de l'Interaction passé en paramètre.

- **+getInteractionByIdTodo:** récupère et renvoie l'adresse de l'Interaction correspondant à un Todo à l'aide de l'id du Todo passé en paramètre.
- **+getTodosByInteractions:** récupère et renvoie dans une liste toutes les adresses des Todo correspondant à chaque Interaction passé en paramètre dans une liste. Utile pour récupérer les Todo d'un Contact.
- **+getInteractionsByTodos:** récupère et renvoie dans une liste toutes les adresses des Interaction correspondant à chaque Todo passé en paramètre dans une liste.
- **+addAssociation :** ajoute une AssociationInteractionTodo à la liste associations.
- **+removeAssociationsByIdInteraction :** supprime de la liste associations toutes les AssociationInteractionTodo correspondant à une Interaction à l'aide de l'id de l'Interaction passé en paramètre.
- **+removeAssociationsByInteractions :** supprime de la liste associations toutes les AssociationInteractionTodo correspondant à toutes les Interaction contenu dans une liste passée en paramètre. Utile lors de la suppression d'un contact.
- **+removeAssociationByIdTodo :** supprime de la liste associations l'AssociationInteractionTodo correspondant à un Todo à l'aide de l'id de Todo passé en paramètre.

10. Classe BDD

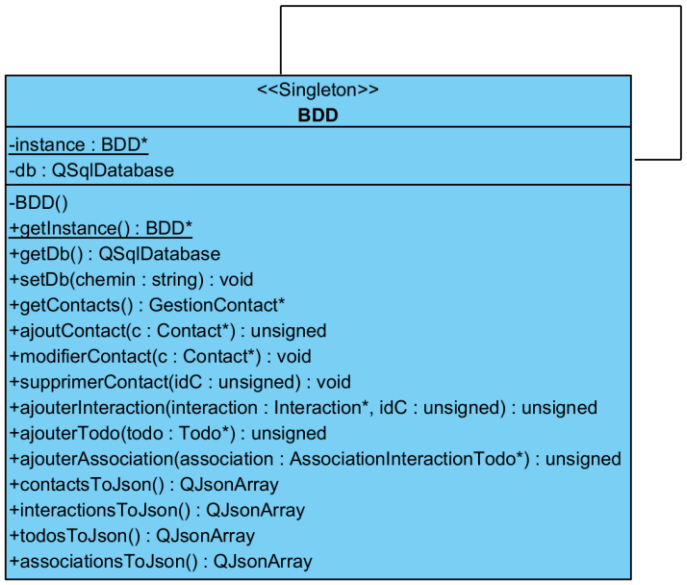

Figure 10 - Classe BDD

Description

Pour pouvoir gérer la base de données, nous avons décidé de créer un singleton qui stocke l'instance QSqlDatabse permettant de communiquer avec la base de données.

Attributs

- **instance :** type BDD*, singleton (static)
- **SQLDatabase db** : stocke le chemin d'une base de données et l'ouvre

Méthodes

- **-BDD :** constructeur par défaut, initialise l'attribut instance. Méthode privée.
- **+getInstance :** getter de l'attribut instance, appel le constructeur si l'attribut instance n'est pas initialisé.
- **+getDb:** getter de l'attribut db. Fonction utiliser dans le main pour savoir si la connexion avec la base de données à bien été effectué (ferme l'application sinon).
- **+setDb:** setter de l'attribut db avec le chemin du fichier sqlite passé en paramètre.
- **+getContacts :** récupère dans la base de données tous les contacts. Pour chaque contact, cherche les interactions. Pour chaque interaction, cherche les todos et associations correspondantes. Puis création d'une Instance contact avec ses interactions dans le gestionnaire d'Interaction, création des Todo et Association correspondants qui sont ajoutés dans le singleton GestionAIT. Les instances Contact créées sont ajoutées à un gestionnaire de contact qui sera retourné à la fin de la fonction.
- **+ajoutContact:** Ajoute dans la BDD le Contact passé en paramètre. Retourne l'id du Contact.
- **+modifierContact :** Modifie dans la BDD le contact passé en paramètre.
- **+supprimerContact:** Supprime dans la BDD le contact ayant l'id passé en paramètre.
- **+ajouterInteraction**: Ajoute dans la BDD l'interaction passé en paramètre appartenant au contact dont l'id est passé en paramètre.
- **+ajouterAssociation**: Ajoute dans la BDD l'Association passé en paramètre. Retourne l'id de l'association.

- **+contactToJson:** Retourne un QJsonArray contenant tous les contacts de la BDD en format Json.
- **+interactionsToJson:** Retourne un QjsonArray contenant toutes les interactions de la BDD en format Json.
- **+todosToJson:** Retourne un QjsonArray contenant tous les todos de la BDD en format Json.
- **+associationsToJson:** Retourne un QjsonArray contenant toutes les associations de la BDD en format Json.

3. Widgets

Dans notre application, toutes les données dans les QtableWidget peuvent être triées grâce à la fonction setSortingEnabled(true) des QtableWidget. La trie s'effectue en cliquant sur une colonne du QtableWidget. La trie se fera en fonction de la colonne sélectionnée. Dans les parties recherches, l'activation du check box de recherche par date décoche le check box de recherche par intervalle (et inversement) car la recherche de ces deux critères simultanément n'est pas possible.

1. Mainwindow

Diagramme de classe du Mainwindow :

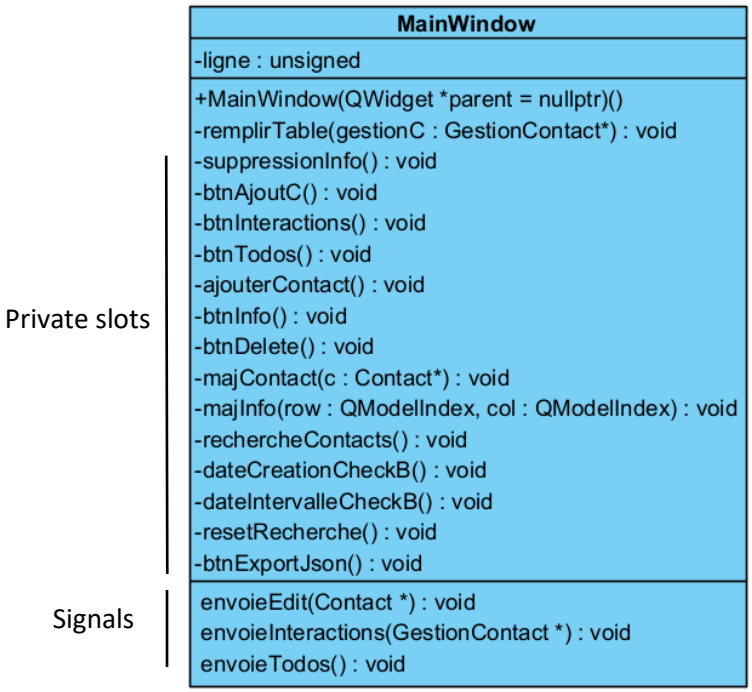

Figure 11 - Classe MainWindow

La fenêtre se présente comme ceci :

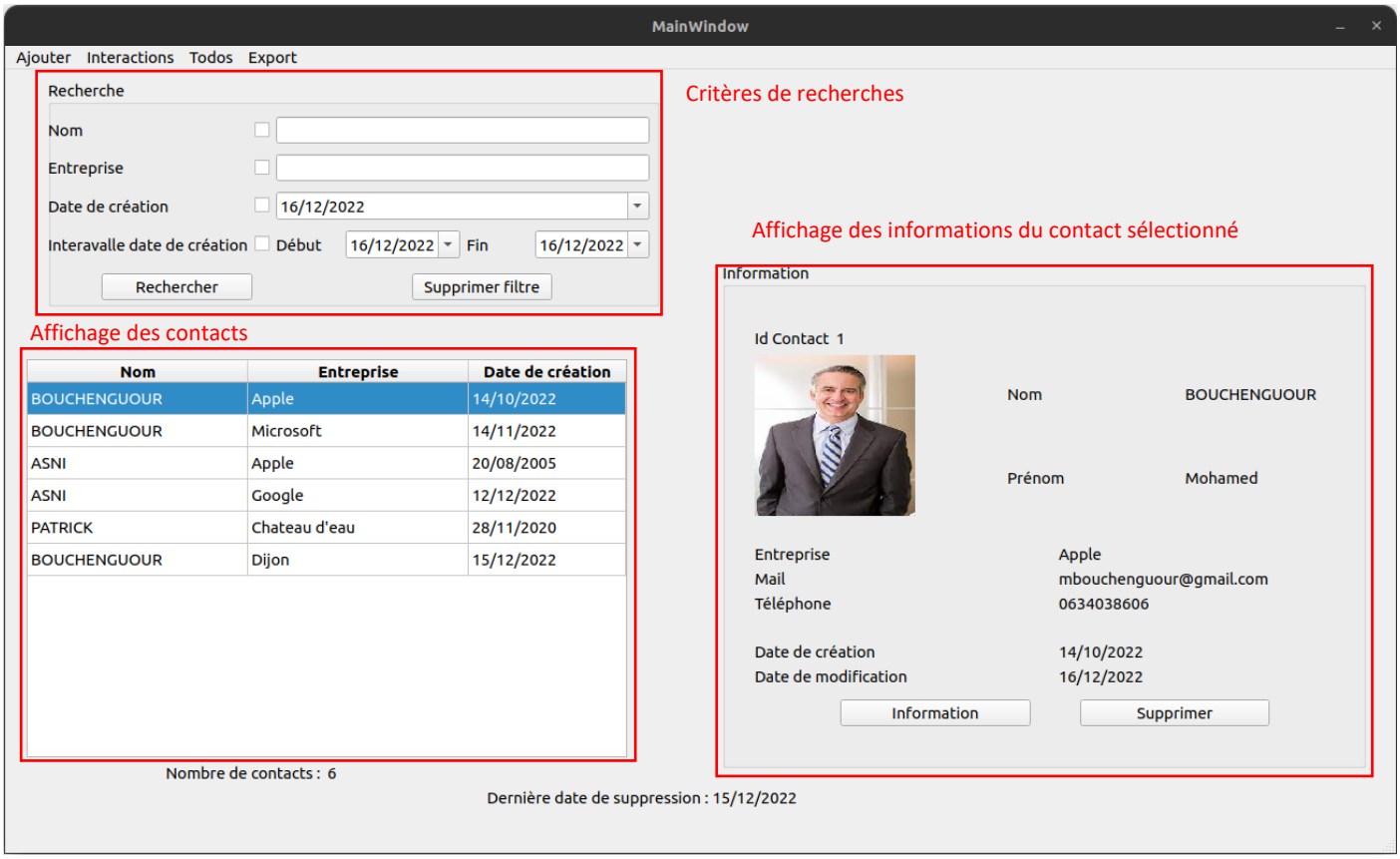

Figure 12 - IHM MainWindow

Ce widget est la fenêtre principale de l'application. Celle-ci nous permet de visualiser la liste des contacts, d'effectuer une recherche sur cette liste, afficher les informations d'un contact, supprimer un contact et accéder au widget de modification d'un contact en fonction du contact sélectionner. Ce widget a également un menu bar qui permet d'accéder au widget d'ajout d'un contact, au widget de recherche des interactions, au widget de recherche de Todos, et permet également d'exporter les données en format json.

Ce widget possède les attributs privés suivants :

- **unsigned ligne** : variable qui stocke la ligne en cours de modification. Cela évite d'effectuer une recherche dans le QtableWidget lorsque l'utilisateur modifie un contact. La modification dans le tableau se fait plus simplement et rapidement.
- **GestionContact * gestionC** : Gestionnaire qui contient la liste de tous les contacts afin de les manipuler et afficher dans le QtableWidget.
- **FormAjoutContact * f** : widget qui affiche un formulaire d'ajout d'un contact.
- **EditContact * edit** : widget qui affiche les informations d'un contact et ses interactions/todos. Permet également de modifier les informations, ajouter une interaction/todo et faire une recherche dans la liste des interactions/todos.
- **RechercheInteractions * rInteractions** : widget qui affiche toutes les interactions de l'application.
- **RechercheTodos * rTodos** : widget qui affiche tous les todos de l'application.

Nous avons décidé de mettre ces widgets dans le MainWindow afin de les connecter plus facilement dans le constructeur du MainWindow.

Fonctions privées :

- **void remplirTable(gestionC : GestionContact *)** : permet de remplir le QTableWidget à partir du gestionnaire de contact mis en paramètre.
- **void suppressionInfo()** : permet de vider les labels affichant les informations d'un contact et désactive les boutons informations et suppression. Cela est nécessaire lorsqu'il y a aucun contact dans le QTbleWidget.

Affichage des contacts :

L'affichage de tous les contacts de l'application se fait à l'aide d'un QTableWidget en affichant seulement le nom, l'entreprise et la date de creation afin de ne pas avoir un tableau trop volumineux. Lorsque l'utilisateur clique sur une ligne, les informations à droite de la fenêtre sont mises à jour.

Les informations du contact sélectionné sont dans des QLabel, eux-mêmes situés dans un groupBox. Il y a deux boutons, le bouton "Information" qui permet d'ouvrir un widget qui affiche les interactions/todos d'un contact mais également des champs de saisies afin de modifier le contact en question. Le bouton "supprimer" permet de supprimer le contact sélectionné. Un QMessageBox s'ouvre afin de demander une confirmation à l'utilisateur.

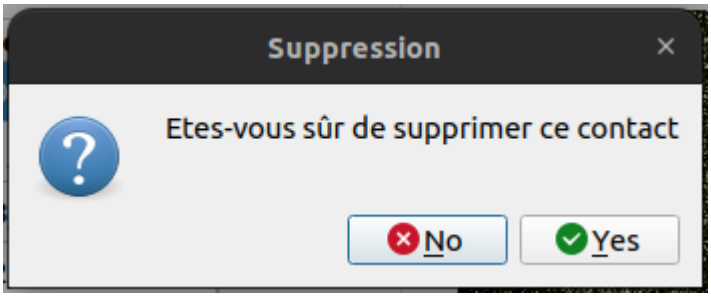

Figure 13 - Confirmation suppression d'un contact

Constructeur :

Tout d'abord on affiche une fenêtre afin de demander à l'utilisateur de choisir une base de données. Si une base de données est sélectionnée, les différents widgets sont initialisés, l'application se connecte à la base de données à l'aide du singleton BDD, récupère la liste de tous les contacts à l'aide de la fonction getContacts et rempli le QTableWidget à partir du gestionnaire de contacts. On initialise les QDateEdit (de la partie recherche) avec la date du jour puis on change la date maximum en affectant la date du jour pour éviter que l'utilisateur saisie une date supérieur à la date du jour. Puis nous réalisons les connexions suivantes :

Le signal "**triggered(bool)**" du bouton "**Ajouter**" du menuBar est connecter au slot "**btnAjoutC()**" du **MainWindow**. Ce slot va ouvrir le widget **FormAjoutContact** qui affiche un formulaire d'ajout de contact.

Le signal "**triggered(bool)**" du bouton "**Interactions**" du menuBar est connecter au slot "**btnInteractions()**" du **MainWindow**. Ce slot va envoyer un signal **envoieInteractions** avec l'attribut gestionC en paramètre.

Le signal "**envoieInteractions(GestionContact*)**" est connecté au slot "**initialiser(GestionContact *)**" de l'attribut **rInteractions** (Widget **RechercheInteractions**). Ce slot va créer un gestionnaire d'interaction, insérer dans le gestionnaire de l'interaction toutes les interactions de l'application à l'aide du gestionnaire de contact du MainWindow, remplir le QTableWidget du widget **RechercheInteractions** à l'aide du gestionnaire de l'interaction puis afficher le widget **RechercheInteractions**.

Le signal "**triggered(bool)**" du bouton "**Todos**" du menuBar est connecter au slot "**btnTodos()**" du MainWindow. Ce slot va envoyer le signal **envoieTodos**.

Le signal "**envoieTodos()**" est connecté au slot "**ouverture()**" de l'attribut **rTodos** (Widget **RechercheTodos**). Ce slot va remplir le QTableWidget contenant les todos à l'aide de la fonction getAllTodos() du singleton GestionAIT puis afficher le widget **RechercheTodos**.

Le signal "**triggered(bool)**" du bouton "**Export**" du menuBar est connecter au slot "**btnExportJson()**" du **MainWindow**. Ce slot va tout d'abord demander à l'utilisateur de sélectionner un dossier. Ce slot va créer dans ce dossier quatre fichiers (contacts, todos, associations, interactions), chacun contenant les informations de la base de données en format JSon à l'aide du singleton BDD.

Le signal "**nouveauContact(Contact *)**" de l'attribut **f** (widget **FormAjoutContact**) est connecté au slot "**ajouterContact(Contact *)**" du **Maindow**. Ce slot ajoute au gestionnaire de contact le contact passé en paramètre du slot. Ensuite, ce nouveau contact est ajouté au QTableWidget contenant les contacts.

Le signal "**clicked()**" du bouton "**Information**" est connecter au slot "**btnInfo()**" du **MainWindow**. Ce slot récupère un contact dans le gestionnaire de contact à partir de la ligne sélectionnée dans le QTableWidget. Il stocke la ligne sélectionné dans l'attribut **ligne** puis envoie le signal "**envoieEdit(Contact *)**" avec le contact récupéré en paramètre.

Le signal "**envoieEdit(Contact *)**" du **MainWindow** est connecté au slot "**initialiser(Contact *)**" de l'attribut **edit** (widget **EditContact**). Ce slot affecte à l'attribut Contact * c du widget **EditContact** le contact passé en paramètre, initialise les QTextEdit contenants les informations du contact, les QTableWidget contenants les interactions/todos du contact puis affiche le widget **EditContact**.

Le signal "**contactModifier(Contact *)**" du widget **EditContact** est connecté au slot "**majContact(Contact *)**" du **MainWindow**. Ce slot met à jour les informations du contact modifié dans le QTableWidget. La ligne du tableau enregistré dans le slot "**btnInfo()**" sera donc modifié.

Le signal "**clicked()**" du bouton "**Supprimer**" est connecter au slot "**btnDelete**" du **MainWindow**. Ce slot va ouvrir un QMessageBox afin de demander une confirmation à l'utilisateur s'il veut supprimer le contact. Si l'utilisateur confirme, le contact est supprimé du gestionnaire de contact, du QTableWidget ainsi que de la base de données. La date de suppression est également mise à jour avec la date du jour.

Le signal "**clicked()**" du bouton "**rechercher**" est connecter au slot "**rechercheContacts()**" du **MainWindow**. Ce slot vérifie la saisie des critères des recherches. Si les critères ne sont pas corrects, on affiche un message d'erreur en dessous du GroupBox "Recherche" avec l'erreur en question (champs vide, critère non sélectionné). Si les critères sont corrects, l'instance GestionContact du **MainWindow** recherche dans sa liste les contacts correspondants aux critères de recherches puis les affiche dans le QTableWidget. L'utilisateur peut sélectionner plusieurs critères de recherche.

Le signal "**clicked()**" du bouton "**Supprimer filtre**" est connecter au slot "**resetRecherche()**" du **MainWindow**. Ce slot supprime les critères de recherches et réaffiche tous les contacts de l'application dans le QTableWidget.

Le signal "**currentRowChanged(QModelIndex,QModelIndex)**" du **QTableWidget** est connecté au slot "**majInfo (QModelIndex, QModelIndex)**" du **MainWindow**. Ce signal se déclenche lorsque l'utilisateur clique sur une ligne du QTableWidget en envoyant la ligne et la colonne correspondante. Le slot met à jour les QLabel contenant les informations du contact en fonction de la ligne sélectionné. SI l'image du contact n'est pas trouvée, une image par défaut (mis en QRessource) est affichée à la place.

Le nombre de contacts est mis à jour à chaque insertion, suppression et recherche (affiche le nombre de contacts trouvés).

2. FormAjoutContact

Diagramme de classe du widget FormAjoutContact :

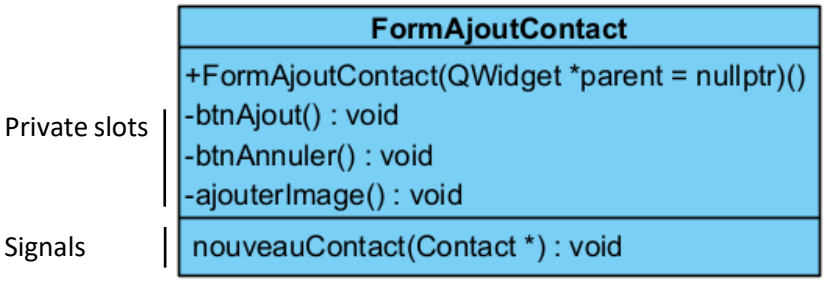

Figure 14 - Classe FormAjoutContact

Ce widget affiche un formulaire d'ajout d'un contact. Il se présente comme ceci :

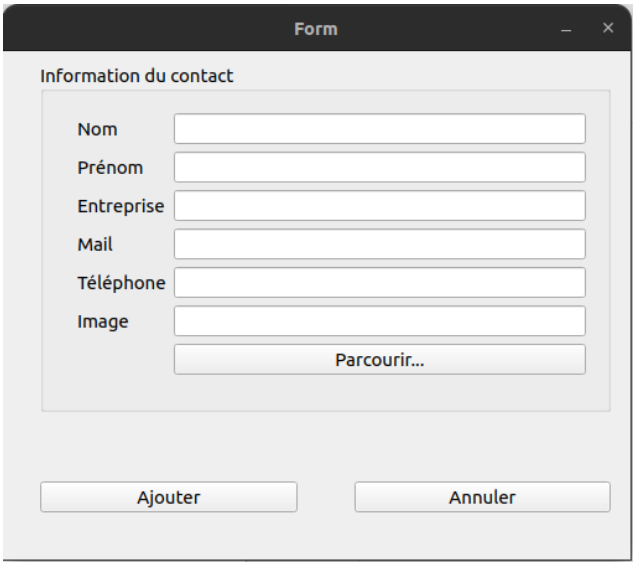

Figure 15 - Formulaire d'ajout d'un contact du widget FormAjoutContact

Le champ de saisie de l'image est en read only pour avoir un chemin valide et éviter que l'utilisateur mette une donnée qui n'est pas un chemin vers une image de type png ou jpg.

Constructeur :

Un validateur est créé pour le QLineEdit du téléphone à l'aide de l'expression régulière "[0-9]{10}". Cette expression empêche l'utilisateur de saisir des caractères et de saisir plus de dix chiffres. Puis nous réalisons les connexions suivantes :

Le signal "**clicked()**" du boutton "**Annuler**" est connecté au slot "**btnAnnuler()**" du widget **FormAjoutContact**. Ce slot vide les champs de saisies et ferme le widget.

Le signal "**clicked()**" du boutton "**parcourir**" est connecté au slot "**ajouterImage()**" du widget **FormAjoutContact**. Ce slot ouvre une fenêtre qui permet à l'utilisateur de choisir une image en format png ou jpg. Le chemin de l'image est affecté aux champs de saisie de l'image.

Le signal "**clicked()**" du boutton "**Ajouter**" est connecté au slot "**btnAjout()**" du widget **FormAjoutContact**. Ce slot vérifie si les champs ne sont pas vides. Il vérifie ensuite la bonne saisie des informations à l'aide d'expression régulière. L'expression régulière "\\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,4}\\b" vérifie si le mail est correct (présence d'un @, d'un point, etc…). L'expression régulière "[0-9]{10}" vérifie si le téléphone contient exactement 10 chiffres. En cas de mauvaise saisie, un QLabel apparait avec l'erreur en question. Si les informations sont correctes, une instance Contact est créé avec les informations saisies, à la date du jour et sans date de modification. Ensuite on appelle, on appelle la fonction ajoutContact du singleton BDD avec le contact crée en paramètre. On modifie l'id du contact avec l'id de la bdd et le widget envoie le signal "**nouveauContact(Contact *)**" avec le contact en paramètre. On rappelle que ce signal sera réceptionné par le **MainWindow** afin d'ajouter le contact dans le QTableWidget contenant les contacts.

3. EditContact

Diagramme de classe :

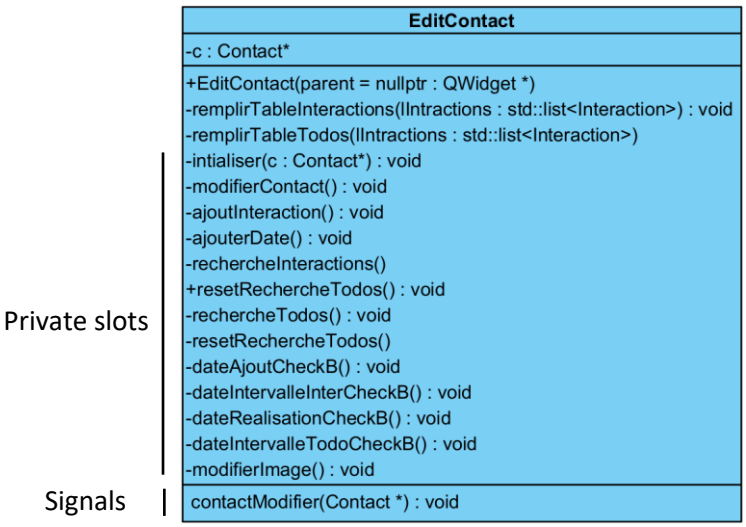

Figure 16 - Classe EditContact

Ce widget se présente comme ceci :

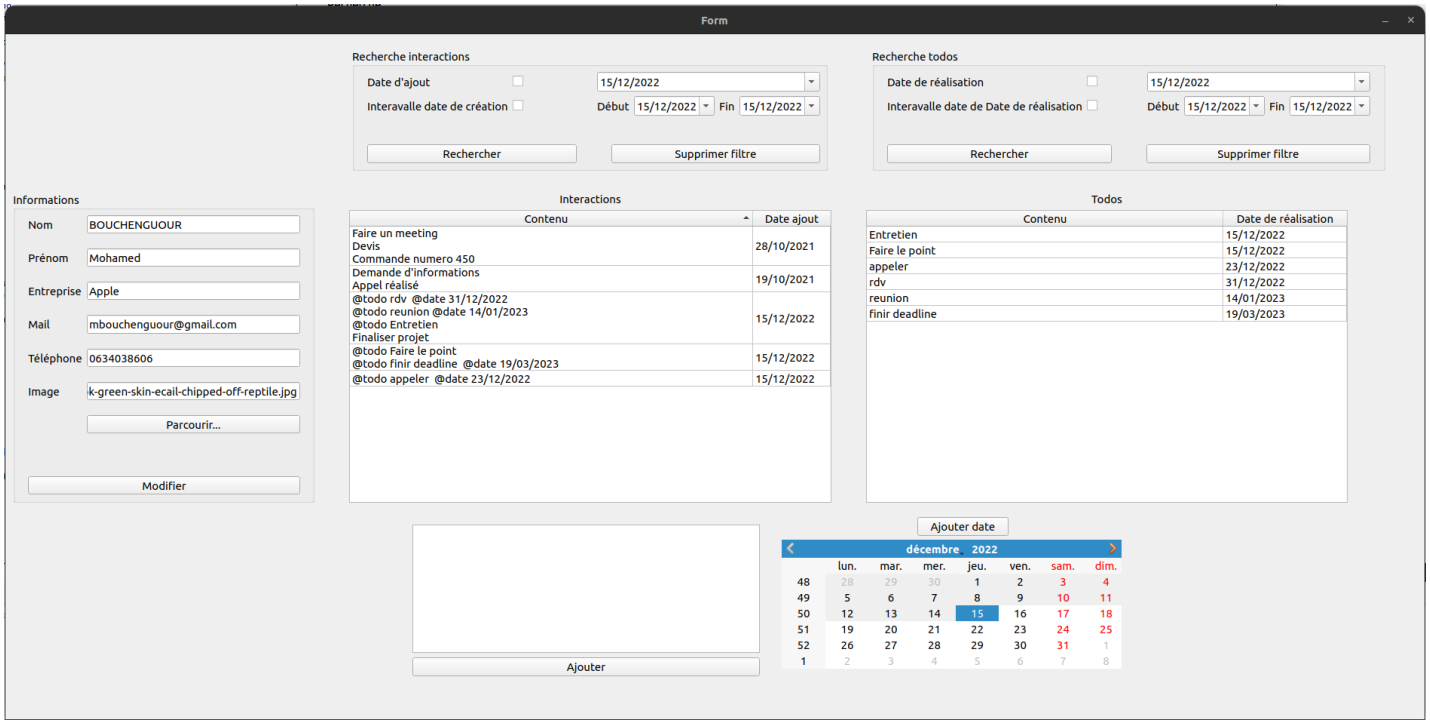

Figure 17 - IHM EditContact

Ce widget affiche les interactions/todos du contact dans des QTableWidget, un QTextEdit afin d'ajouter une interaction et un QCalendarWidget afin d'ajouter un tag date lors de l'ajout d'une nouvelle interaction. Il affiche également les informations modifiables (nom, prénom, entreprise, mail, téléphone et l'image) dans des QLineEdit afin de permettre à l'utilisateur de modifier les informations. En haut de chaque QTableWidget l'utilisateur peut saisir des critères de recherche (interactions ou todos).

Ce widget possède un attribut **c** de type Contact afin de manipuler les informations plus facilement dans n'importe quelle fonction.

Ce widget possède également deux fonctions privées :

- **-remplirTableInteractions(lInteractions : std::list<Interaction>)** : remplie le QTableWidget des Interactions à partir d'une liste d'Interaction.
- **-remplirTableTodos(lTodos : std::list<Todo *>)** : remplie le QTableWidget des Todos à partir d'une liste de pointeur de Todo.

Constructeur :

Un validateur est créé pour le QLineEdit du téléphone à l'aide de l'expression régulière "[0-9]{10}". Cette expression empêche l'utilisateur de saisir des caractères et de saisir plus de dix chiffres. Puis nous réalisons les connexions suivantes :

Le signal "**clicked()**" du boutton "**Modifier**" est connecté au slot "**modifierContact()**" du widget **EditContact**. Ce slot est très similaire au slot "**btnAjout()**" du widget **FormAjoutContact**. Ce slot vérifie si les champs ne sont pas vides, la bonne saisie des informations avec les mêmes expressions régulières et affiche les mêmes messages d'erreur. Si les données saisies sont correctes, l'attribut Contact * c est modifié à l'aide des champs de saisies. Puis, on appelle la fonction **modifierContact** du singleton BDD afin de modifier le contact dans la base de données. Le signal "**contactModifier(Contact *)**" est ensuite envoyé avec le contact en paramètre. On rappelle que ce signal est réceptionné par le **MainWindow** afin de mettre à jour le contact modifié dans le QTableWidget. Un QMessageBox est ensuite affiché pour informer à l'utilisateur que le contact est modifié.

Le signal "**clicked()**" du boutton "**parcourir**" est connecté au slot "**modifierImage()**" du widget **EditContact**. Ce slot ouvre une fenêtre qui permet à l'utilisateur de choisir une image en format png ou jpg. Le chemin de l'image est affecté aux champs de saisie de l'image.

Le signal "**clicked()**" du boutton "**Ajouter date**" est connecté au slot "**ajouterDate()**" du widget **EditContact**. Ce slot récupère la ligne du curseur du clavier dans le QTextEdit. Il vérifie si la ligne possède un tag @todo en début de ligne à l'aide l'expression régulière "^@todo .*$". Si la ligne n'a pas de tag @todo, le message d'erreur "Vous ne pouvez pas ajouter de date s'il n'y a pas de @todo" est affiché dans un QLabel. Il vérifie ensuite si un tag @date est présent dans la ligne à l'aide l'expression régulière " @date ". Si la ligne a déjà un tag @date, le message d'erreur "Il y a déjà une date pour ce todo " est affiché dans un QLabel. S'il y a un tag @todo mais de tag @date dans la ligne, le slot récupère la date choisis dans le QCalendarWidget et ajoute un tag @date avec la date sélectionnée à la fin de la ligne.

Le signal "**clicked()**" du boutton "**Ajouter**" est connecté au slot "**ajoutInteraction()**" du widget **EditContact**. Ce slot récupère le contenu du QTextEdit et cherche dans chaque ligne la présence d'un tag @todo à l'aide de l'expression régulière "^@todo .*$". Il stocke chaque ligne contenant un tag @todo dans un QStringList. Il vérifie ensuite dans chaque ligne contenant un tag @todo s'il y a un tag @date. Si un tag @date est présent, il vérifie si c'est l'avant dernier mot de la ligne. Si le tag @date n'est pas à la bonne place, le slot affiche dans un QLabel le message d'erreur "Date du todo incorrect, veuillez saisir la date à la fin de la ligne". Si l'emplacement du tag @date est correcte, on récupère le dernier mot de la ligne (emplacement de la date) et vérifie si la date est en format "jj/mm/aaaa" à l'aide de l'expression régulière "^(\\d\\d)/(\\d\\d)/(\\d\\d\\d\\d)$". Si le format de la date n'est pas

correct, le message d'erreur "Le format de la date doit être "JJ/MM/AAAA"" est affiché dans un QLabel. Si le format de la date est correct, on vérifie si la date est inférieure à la date du jour à l'aide de la classe GestionDate. Si la date est inférieure à la date du jour, le message d'erreur "La date du todo ne peux pas être avant la date du jour" est affiché dans un Qlabel. Si on a un message d'erreur, l'insertion ne se fait pas. Ensuite, si la saisie est correcte, on crée une interaction avec le contenu du QTextEdit et la date du jour. On ajoute cette interaction dans la base de données à l'aide de la fonction ajouterInteraction du singleton BDD. On ajoute l'interaction dans le QTableWidget des Interaction. Puis on recherche dans chaque ligne du QTextEdit la présence d'un tag @todo. Si un tag @todo est présent, on cherche la présence d'un tag date @date. Si une date est présente, on crée une instance GestionDate avec. Sinon, on crée l'instance GestionDate avec la date du jour. On crée ensuite une instance Todo avec le contenu de la ligne sans les 6 premières caractères (on enlève le @todo ␣) et sans les 2 derniers mots si un tag @date est présent (tag @date + la date). On créer une Instance AssociationInteractionTodo avec l'interaction et le todo. On ajoute le todo, l'interaction puis l'association dans la base de données. On ajoute ensuite le Todo dans le QTableWidget contenant les Todo.

Le signal "**clicked()**" bouton "**Rechercher**" dans le GroupBox "Recherche interactions" est connecté au slot "**rechercheInteractions()**". Ce slot vérifie la saisie des critères. Il vérifie si un checkBox est sélectioné. Si l'utilisateur lance une recherche sans critère, le message d'erreur "Veuillez choisir un critère" est affiché dans un QLabel.

Pour la recherche par date, on cherche dans le gestionnaire des interactions du contact les Interaction crées à une date donnée à l'aide de la fonction getInteractionsByDate avec en paramètre une instance GestionDate créée à l'aide du QDateEdit.

Pour la recherche par intervalle, on vérifie d'abord si la date de début est inférieure à la date de fin. Si ce n'est pas le cas, le message d'erreur "La date de début doit être inférieur à la date de fin" est affiché dans un QLabel. On cherche dans le gestionnaire des interactions du contact, les Interaction créées dans un intervalle de date à l'aide de la fonction getInteractionsByIntervalle avec en paramètre les deux instances dates crées à partir des QDateEdit.

On stocke ces interactions dans une instance GestionInteraction temporaire. On remplie ensuite la table des Interaction et la table des Todo correspondant (à l'aide de la fonction getTodos du gestionnaire d'Interaction).

Le signal "**clicked()**" bouton "**Rechercher**" dans le GroupBox "Recherche Todos" est connecté au slot "**rechercheTodos()**". Ce slot fonctionne comme le slot "**rechercheInteraction()**" mais nous cherchons d'abord les Todo correspondants aux critères à l'aide de liste des Todo du contact (récupérée grâce à la fonction getTodos du gestionnaire d'Interaction). Ensuite, nous cherchons les Interaction correspondant à ces Todo à l'aide de la fonction getInteractionsByTodos du singleton GestionAIT en passant la liste des Todo trouvée en paramètre.

Le signal "**clicked()**" des boutons "**Supprimer Filtre**" des Interaction et des Todo sont connectés aux slots "**resetRechercheInter()**" et "**resetRechercheTodo()**". Ces slots suppriment les critères de recherche et réaffiche toutes les Interaction/Todo du contact.

4. RechercheInteractions :

Diagramme de classe du widget RechercheInteractions :

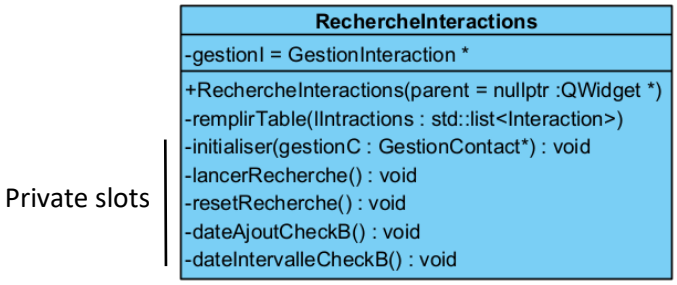

Figure 18 - Classe RechercheInteractions

Ce widget se présente comme ceci :

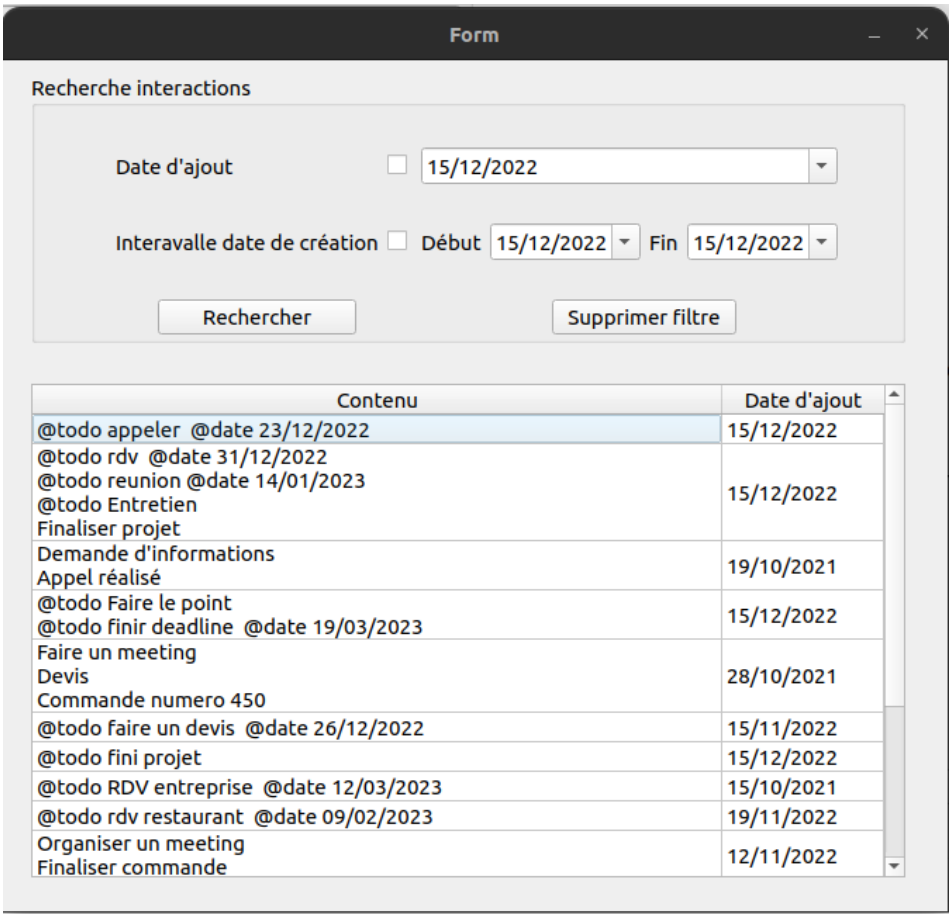

Figure 19 - IHM RechercheInteractions

Ce widget affiche toutes les interactions de l'application. Il permet également d'effectuer une recherche sur cette liste.

Ce widget à un attribut **gestionI** de type GestionInteraction. Celui-ci permet de stocker toutes les interactions de l'application. Il est initialisé lors du déclenchement du slot "**initialiser(GestionContact*)**" lors de la réception du signal "**envoieInteractions(gestionContact*)**" du **MainWindow**. Ce slot récupère toutes les interactions des contacts du gestionnaire de contact passé en paramètre qu'il ajoute à l'attribut **gestionI**. Il remplit le QTableWidget à partir de **gestionI** puis affiche le widget.

Ce widget à une fonction privée :

- -**remplirTable(lInteractions : std::list<Interaction>)** qui remplit le QTableWidget à partir de la liste d'Interaction mis en paramètre. Cette fonction est privée.

Constructeur :

Le signal "**clicked()**" du bouton "**Rechercher**" est connecté au slot "**lancerRecherche()**" du widget **RechercheInteractions**. Ce slot vérifie si un checkBox est sélectionné. Si l'utilisateur lance une recherche sans critère, le message d'erreur "Veuillez choisir un critère" est affiché dans un QLabel.

Pour la recherche par date, le slot cherche dans **gestionI** les Interaction crées à une date donnée à l'aide de la fonction getInteractionsByDate avec en paramètre une instance GestionDate créée à l'aide du QDateEdit.

Pour la recherche par intervalle, on vérifie d'abord si la date de début est inférieure à la date de fin. Si ce n'est pas le cas, le message d'erreur "La date de début doit être inférieur à la date de fin" est affiché dans un QLabel. On cherche dans **gestionI** les Interaction crées dans un intervalle de date à l'aide de la fonction getInteractionsByIntervalle avec en paramètre les deux instances dates crées à partir des QDateEdit.

On stocke ces interactions dans une instance GestionInteraction temporaire. On remplit ensuite la table des Interaction à l'aide de cette instance.

Le signal "**clicked()**" bouton "**Supprimer Filtre**" est connecté au slot "**resetRecherche()**" du widget **RechercheInteractions**. Ce slot supprime les critères de recherche et affiche toutes les interactions de l'application dans le QTableWidget.

5. RechercheTodos :

Diagramme de classe :

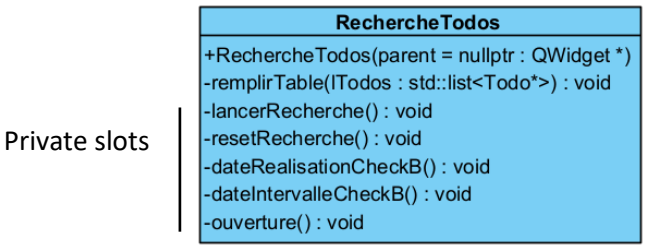

Figure 20 - Classe RechercheTodos

Ce widget se présente comme ceci :

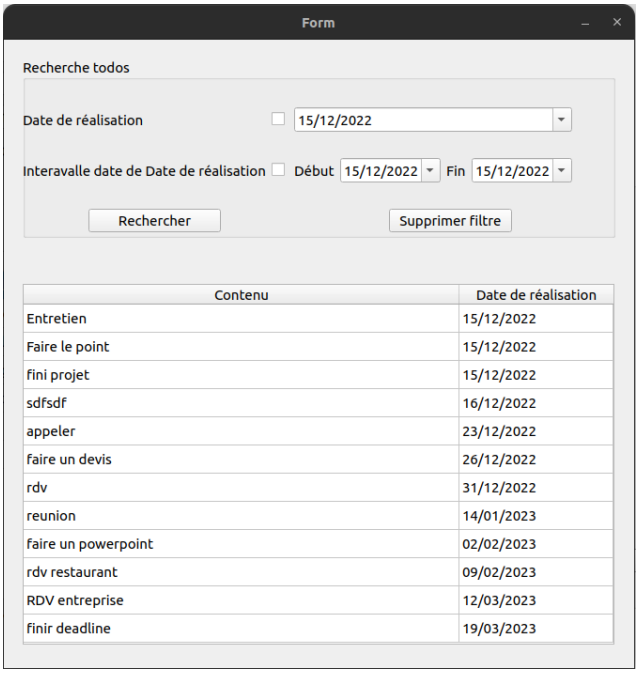

Figure 21 - IHM RechercheTodos

Ce widget à une fonction privée :

- **-remplirTable(lTodos : std::list<Todo *>)** qui remplit le QTableWidget à partir de la liste de pointeur de Todo mis en paramètre. Cette fonction est privée.

Le signal "**clicked()**" du bouton "**Rechercher**" est connecté au slot "**lancerRecherche()**" du widget **RechercheTodos**. Ce slot vérifie si un checkBox est sélectionné. Si l'utilisateur lance une recherche sans critère, le message d'erreur "Veuillez choisir un critère" est affiché dans un QLabel.

Pour la recherche par date, le slot cherche dans la liste de tous les Todo de l'application (récupérer grâce à la fonction getAllTodos du singleton GestionAIT) les Todo dont la date de réalisation correspond à la date saisie par l'utilisateur dans le QDateEdit.

Pour la recherche par intervalle, on vérifie d'abord si la date de début est inférieure à la date de fin. Si ce n'est pas le cas, le message d'erreur "La date de début doit être inférieur à la date de fin" est affiché

dans un QLabel. Le slot cherche dans la liste de tous les Todo de l'application (récupérer grâce à la fonction getAllTodos du singleton GestionAIT) les Todo dont la date de réalisation est comprise entre la date de début et la date de fin saisie par l'utilisateur dans les QDateEdit correspondant.

Le signal "**clicked()**" bouton "**Supprimer Filtre**" est connecté au slot "**resetRecherche()**" du widget **RechercheTodos**. Ce slot supprime les critères de recherche et affiche tous les Todo de l'application dans le QTableWidget.

6. SQL

Diagramme de la base de données

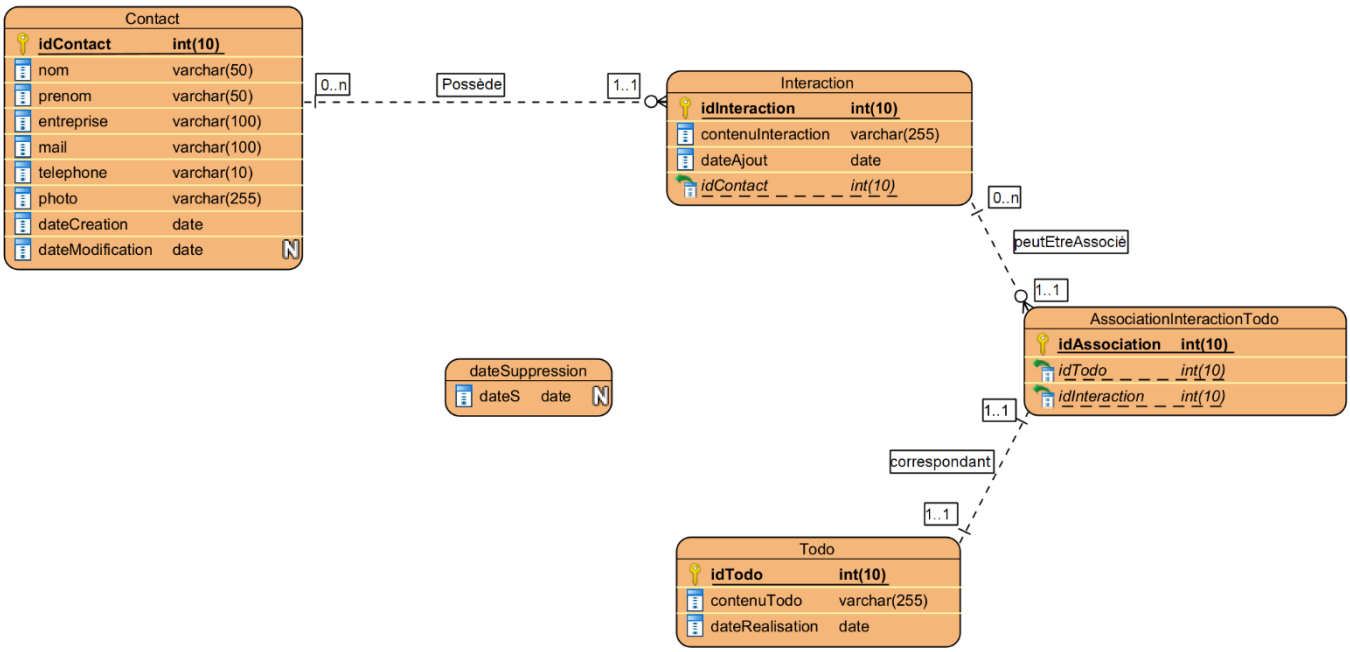

Figure 22 - Diagramme BDD

**Cardinalités avec Merise*

Tables :

Table Contact :

- idContact (clé primaire) : attribut de type INTEGER qui stocke l'id du Contact
- nom: attribut de type VARCHAR2(50) qui stocke le nom du Contact
- prenom: attribut de type VARCHAR2(50) qui stocke le prénom du Contact
- entreprise: attribut de type VARCHAR2(100) qui stocke l'entreprise du Contact
- mail: attribut de type VARCHAR2(100) qui stocke le mail du Contact
- telephone : attribut de type VARCHAR2(10) qui stocke le numéro de téléphone du Contact. Donnée de taille 10 car un téléphone contient 10 chiffres. L'attribut n'est pas de type INTEGER car on ne peut pas réaliser de traitement avec un numéro de téléphone (somme, différence, etc…)
- photo : attribut de type VARCHAR2(255) qui stocke l'URI de la photo du Contact. Donnée de taille 255 au cas où l'URI est longue.
- dateCreation : attribut de type DATE.
- dateModification : attribut de type DATE. Cet attribut peut être NULL si le contact n'a pas toujours été modifié.

Table Interaction :

- idInteraction (clé primaire) : attribut de type INTEGER qui stocke l'id de l'interaction.
- contenuInteraction: attribut de type VARCHAR2(255) qui stocke le contenu d'une interaction
- dateAjout : attribut de type DATE qui stocke la date d'ajout de l'Interaction.
- idContact (clé étrangère correspondant à un idContact de la table Contact) : attribut de type INTEGER qui stocke l'id d'un contact. Permet de lier une interaction à un contact.

Table Todo :

- idTodo (clé primaire) : attribut de type INTEGER qui stocke l'id d'un Todo.
- contenuTodo: attribut de type VARCHAR2(255) qui stocke le contenu Todo.
- dateRealisation : attribut de type DATE qui stocke la date de réalisation d'un Todo.

Comme il n'y a pas de clé étrangère dans cette table, la suppression des todo après la suppression d'un contact ne se fera pas automatiquement. Pour ceci, nous avons implémenter le trigger suivant qui supprime les Todo correspondant à un contact qui vient d'être supprimé.

```
CREATE TRIGGER IF NOT EXISTS deleteTodo
AFTER DELETE ON AssociationInteractionTodo
BEGIN
    DELETE FROM Todo WHERE Todo.idTodo = old.idTodo;
END;
```

Figure 23 - Trigger deleteTodo

Table AssociationInteractionTodo :

- idAssociation (clé primaire) : attribut de type INTEGER qui stocke l'id de l'association entre une interaction et un todo.
- idTodo (clé étrangère correspondant à un idTodo de la table Todo) : attribut de type INTEGER qui stocke l'id d'un todo.
- idInteraction (clé étrangère correspondant à un idInteraction de la table Interaction): attribut de type INTEGER qui stocke l'id d'une interaction.

Table dateSuppression :

Cette table stocke un attribut de type Date qui correspondant à la dernière date de suppression d'un contact. Cette table stocke une seule ligne qui sera mis à jour après chaque suppression d'un contact à l'aide du trigger suivant :

```
CREATE TRIGGER IF NOT EXISTS updateDateSuppression
AFTER DELETE ON Contact
BEGIN
    UPDATE dateSuppression SET dateS= date();
END;
```

Figure 24 - Trigger updateDateSuppression

Script de la base de données

```
DROP TABLE IF EXISTS AssociationInteractionTodo;
DROP TABLE IF EXISTS Todo;
DROP TABLE IF EXISTS Interaction;
DROP TABLE IF EXISTS Contact;

DROP TABLE IF EXISTS dateSuppression;
CREATE TABLE dateSuppression(dateS);
INSERT INTO dateSuppression VALUES (NULL);

CREATE TABLE Contact (
  idContact        INTEGER PRIMARY KEY AUTOINCREMENT,
  nom              VARCHAR2(50) NOT NULL,
  prenom           VARCHAR2(50) NOT NULL,
  entreprise       VARCHAR2(100) NOT NULL,
  mail             VARCHAR2(100) NOT NULL,
  telephone        VARCHAR2(10) NOT NULL,
  photo            VARCHAR2(255) NOT NULL,
  dateCreation     DATE NOT NULL,
  dateModification DATE
  );

CREATE TABLE Interaction (
  idInteraction      INTEGER PRIMARY KEY AUTOINCREMENT,
  contenuInteraction VARCHAR2(255) NOT NULL,
  dateAjout          DATE NOT NULL,
  idContact          INTEGER NOT NULL,
  FOREIGN KEY (idContact) REFERENCES Contact(idContact) ON DELETE CASCADE
  );

CREATE TABLE Todo (
  idTodo          INTEGER PRIMARY KEY AUTOINCREMENT,
  contenuTodo     VARCHAR2(255) NOT NULL,
  dateRealisation DATE NOT NULL
  );

CREATE TABLE AssociationInteractionTodo (
  idAssociation INTEGER PRIMARY KEY AUTOINCREMENT,
  idTodo        INTEGER NOT NULL,
  idInteraction INTEGER NOT NULL,
  FOREIGN KEY (idTodo) REFERENCES Todo (idTodo) ON DELETE CASCADE,
  FOREIGN KEY (idInteraction) REFERENCES Interaction (idInteraction) ON DELETE CASCADE
  );
```

Figure 25 - Script de la base de données

Les clés étrangères ont des ON DELETE CASCADE. Cela permet après la suppression d'un contact de supprimer les interactions de ce contact, puis les AssociationInteractionTodo correspondant à ces interactions. Les todos seront supprimés à l'aide du trigger.