

Projet de CDAA Logiciel de Gestion de relations clients

Introduction : Dans ce projet de Conception et Développement Avancé d'Applications, nous avons créé un logiciel de gestion de relations clients permettant d'enregistrer des fiches de contacts, des interactions pouvant contenir des tâches à faire extraites automatiquement. Ces tâches peuvent être marquées comme faites (ou à faire). Il est également possible de rechercher des contacts, modifier leur fiche, ajouter des interactions (nouvelles ou existantes sur un autre contact au cas où plusieurs contacts aient pris part à une même interaction), afficher la liste des prochaines choses à faire ou des interactions s'étant produites entre 2 dates. Toutes les données utilisées sont sauvegardées dans une base de données (SQLite) mais elles sont également exportables/importables via le format JSON.

Nous allons présenter dans ce rapport les différentes parties de notre logiciel : les fenêtres de la partie graphismes, la programmation orientée objets dans la gestion interne des instances et la persistance des données dans la base de données ainsi que les connexions entre les différentes parties.

I. Graphismes

La première partie que nous allons évoquer est l'interface avec l'utilisateur : les fenêtres (ou widgets). Il s'agit de QMainWindow (pour la fenêtre principale) et de QWidget créés grâce au QT Designer. L'emploi à des QDialog peut également avoir lieu pour demander des confirmations à l'utilisateur par exemple.

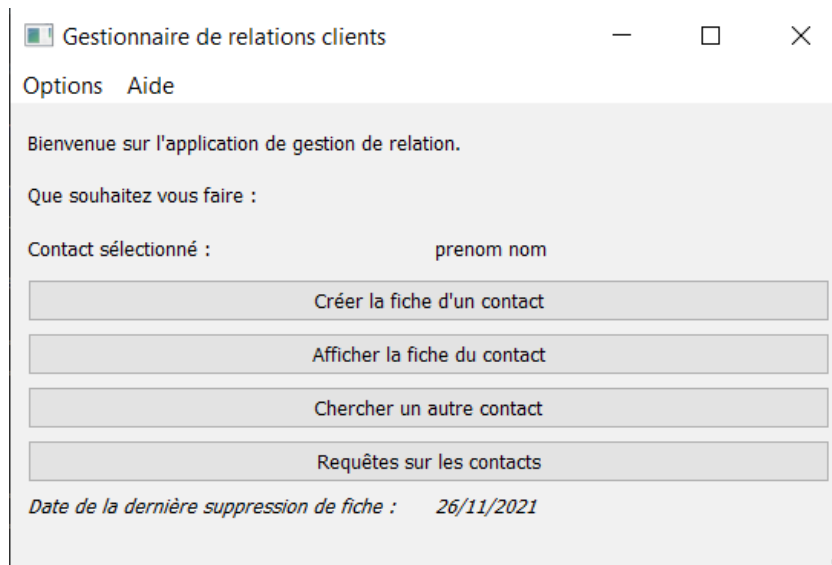
1. Fenêtre principale

Lorsque l'utilisateur démarre le logiciel, la fenêtre principale est affichée. Il s'agit d'une fenêtre faisant office de « menu principal » centralisant les boutons pour ouvrir les autres fenêtres et permettant d'utiliser des options comme exporter/importer des données via le format JSON ou quitter l'application. On peut aussi lire des informations comme le contact actuellement sélectionné pour toute l'application ou la date de la dernière suppression de fiche.



Visuel de la fenêtre principale

Cette fenêtre est gérée par une classe « MainWindow » héritant de QMainWindow. Elle crée d'autres fenêtres (correspondant aux boutons) initialement cachées et les affiche lors du click sur le bouton correspondant à la fenêtre. Le bouton pour afficher la fiche du contact est désactivé si aucun contact n'est sélectionné (au lancement de l'application ou après la suppression de la fiche du contact qui avait été sélectionné).



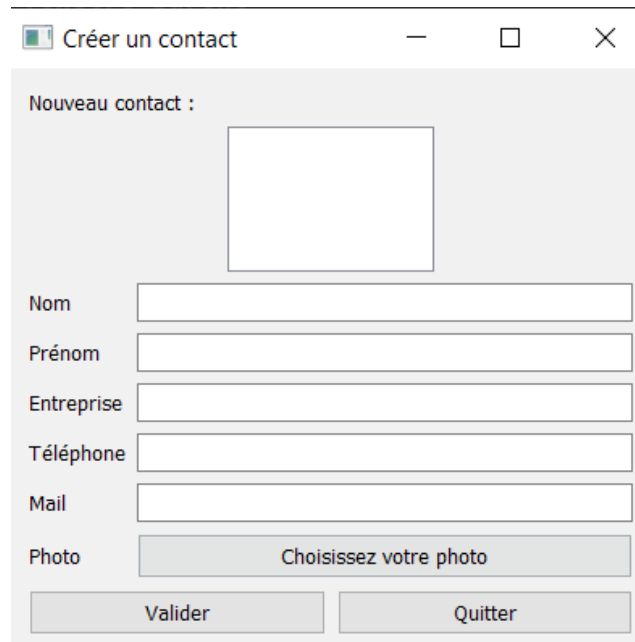
Visuel de la fenêtre principale lorsqu'un contact est sélectionné

En plus d'autres fenêtres (QWidgets), la MainWindow se sert de QDialog et permet également d'afficher un message « A propos » lors de la sélection de l'option du même nom par la biais d'une QDialog de sous type QMessageBox. Il est également permis à l'utilisateur de sélectionner un fichier pour importer/exporter les données du logiciel grâce à des QDialog de type QFileDialog : getSaveFileName pour sauvegarder sur un fichier ou getOpenFileName pour importer depuis un fichier déjà existant.

Puisqu'il s'agit de la fenêtre centrale pour toute l'application, nous avons décidé que la fermeture de celle ci doit fermer tout le programme et pour cela nous avons surchargé l'évènement de fermeture de cette fenêtre (closeEvent) pour également rajouter la fermeture de l'application entière via QApplication::quit().

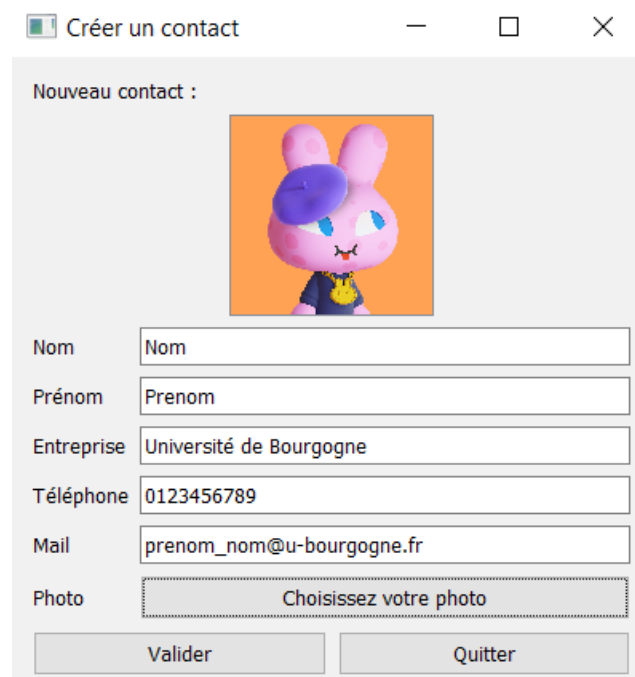
2. Fenêtre de création de contact

Lors de la rencontre d'un nouveau contact, l'utilisateur va généralement créer une fiche avec ses informations. Il va donc cliquer sur le bouton « créer la fiche d'un contact » de la fenêtre principale qui va alors ouvrir la fenêtre permettant la création d'une nouvelle fiche de contact.



Visuel de la fenêtre de création de contact

Sur cette fenêtre simple, gérée par une classe « CreationFicheWindow » héritant de Qwidget, il est possible d'entrer diverses informations textuelles (nom, prénom, entreprise, mail, téléphone) et de sélectionner une photo à afficher. On peut également voir un aperçu de l'image sélectionnée.



*Visuel de la fenêtre de création de contact
remplie avec des informations et une photo sélectionnée*

Lors du click sur le bouton pour choisir une photo, une QDialog est utilisée par le widget : une QFileDialog est affichée dans le but de sélectionner l'image (getOpenFileName). Un filtre est appliqué sur les fichiers afin de n'afficher que les images de type png ou jpg : « PNG File (*.png) ;; JPG File (*.jpg) ».

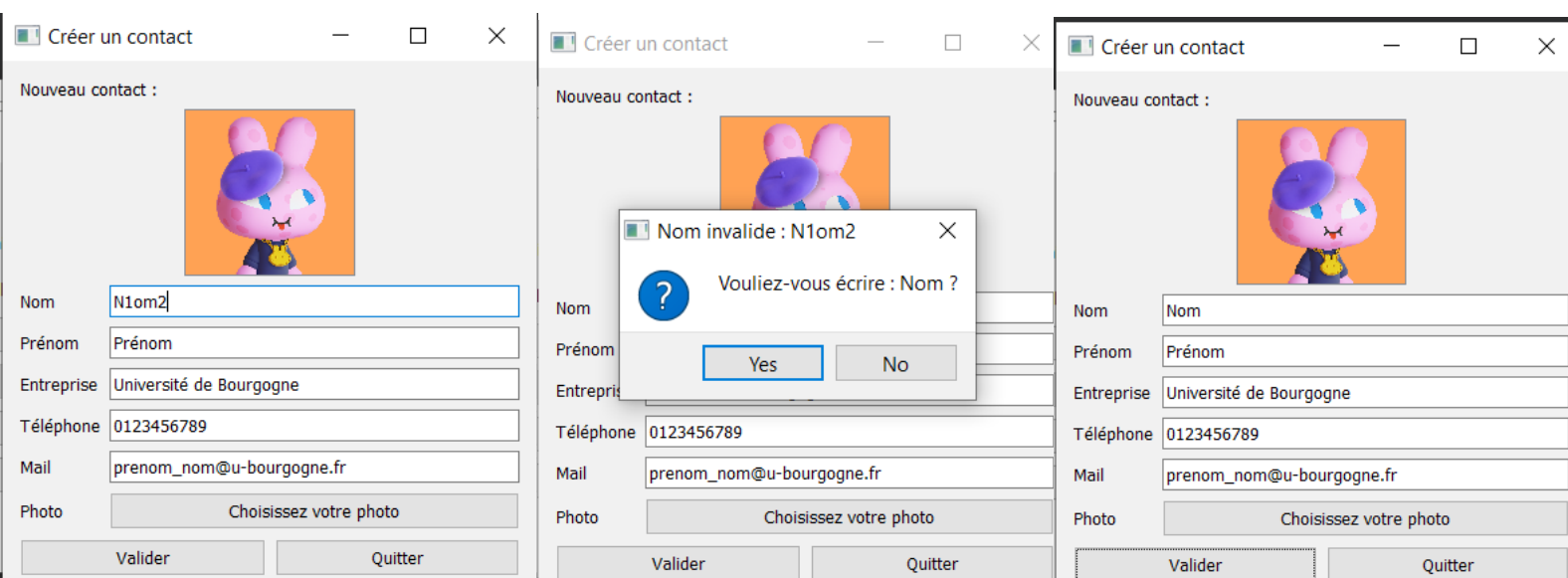
Une fois le fichier sélectionnée par l'intermédiaire de cette boîte de dialogue, l'image est chargée dans une `QGraphicsScene`, via `addPixmap(QPixmap::fromImage(QImage))`. La `QImage` en question est obtenue en utilisant le chemin du fichier de l'image dans le constructeur de `QImage` puis en la redimensionnant aux dimensions du rectangle d'affichage (`QGraphicsView`) via `QImage.scaled(dimensions)`. On peut enfin afficher la scène dans la `QGraphicsView` en utilisant `setScene`. Dans le cas où l'image n'aurait pas réussi à être chargée ou que le chemin aurait été vide, on fera un `setScene(nullptr)` afin de ne rien afficher.

Cependant, il faut également réafficher l'image lorsque le cadre est redimensionné, ce qui se produit lors d'un changement des dimensions de la fenêtre par l'utilisateur. Pour cela, nous avons surchargé l'évènement de redimensionnement de la fenêtre (`resizeEvent`). Il suffit alors de tout réafficher comme si le chemin de l'image venait d'être sélectionné, via la même méthode d'affichage.

Une fois les informations dont l'utilisateur dispose sont entrées sur le nouveau contact, il peut alors créer la fiche en appuyant sur le bouton valider. Les informations sont alors collectées et des vérifications sont effectuées pour essayer de stopper les erreurs d'écritures : on va tester si le nom et le prénom ne sont pas vides et contiennent uniquement des lettres, si le numéro de téléphone est bien constitué de 10 chiffres, si le mail a le bon format, ... Dans le cas où une incohérence est détectée, nous allons utiliser des boîtes de dialogue pour avertir l'utilisateur : des `QMessageBox` vont afficher des messages d'erreur (`::warning`).

Pour le nom et le prénom, nous avons développés des fonctions permettant de corriger une entrée incohérente pour proposer à l'utilisateur une autre version. On va alors proposer à l'utilisateur de plutôt saisir automatiquement la version corrigée via une question dans une boîte de dialogue : `QMessageBox::question`. Cette fonction va renvoyer le bouton qu'a utilisé l'utilisateur sur la `QMessageBox` et nous avons plus qu'à tester si ce bouton est égal au bouton « oui » (`QMessageBox::Yes`) pour appliquer automatiquement les changements.

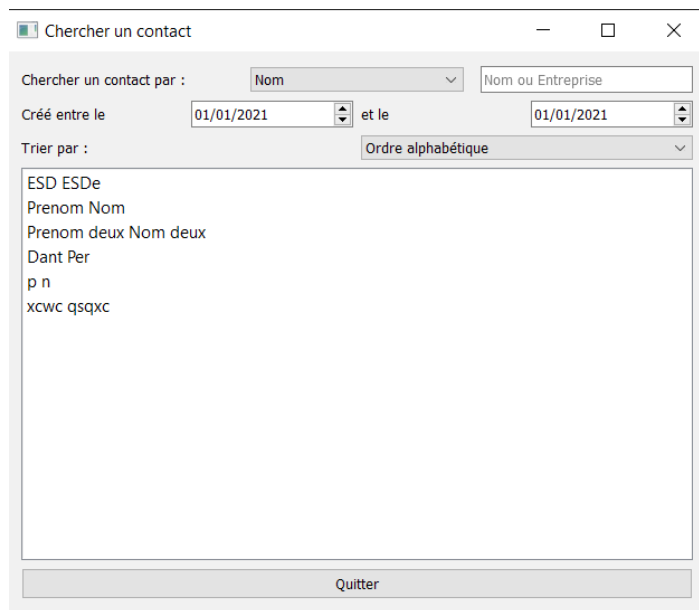
Si toutes les données entrées sont jugées cohérentes, le nouveau contact sera créé et la fenêtre se fermera.



Proposition d'un nom plus cohérent en cas de faute de frappe

3. Fenêtre de recherche

L'utilisateur peut avoir besoin de rechercher un contact à sélectionner dans le but d'effectuer divers opérations que nous détaillerons plus loin. Il pourra alors utiliser la fenêtre de recherche, accessible depuis la fenêtre principale (mais pas seulement) dans le but de sélectionner le contact « courant » pour toute l'application. Cette fenêtre permet de trier tous les contacts par ordre alphabétique ou par date de création et de filtrer par nom/prénom ou entreprise donnés parmi les fiches créées dans l'intervalle de deux dates, puis de cliquer sur un élément de la liste affichée pour le sélectionner.



Chercher un contact

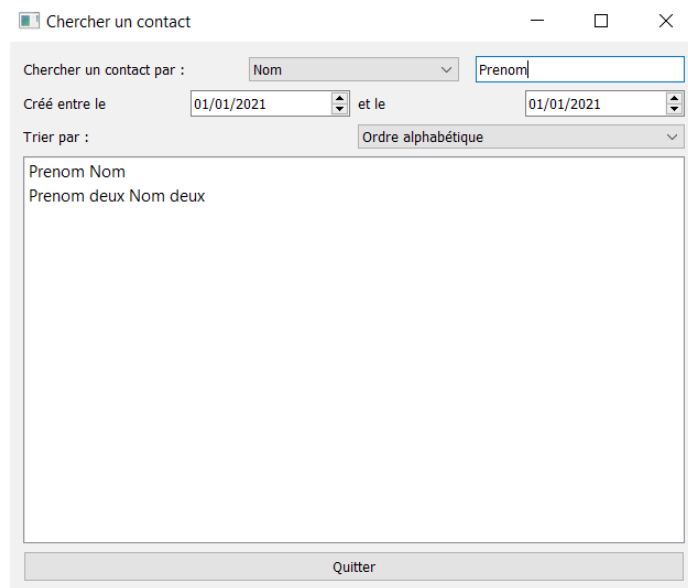
Chercher un contact par : **Nom**

Créé entre le et le

Trier par : **Ordre alphabétique**

- ESD ESDe
- Prenom Nom
- Prenom deux Nom deux
- Dant Per
- p n
- xcwc qsqxc

Quitter



Chercher un contact

Chercher un contact par : **Nom**

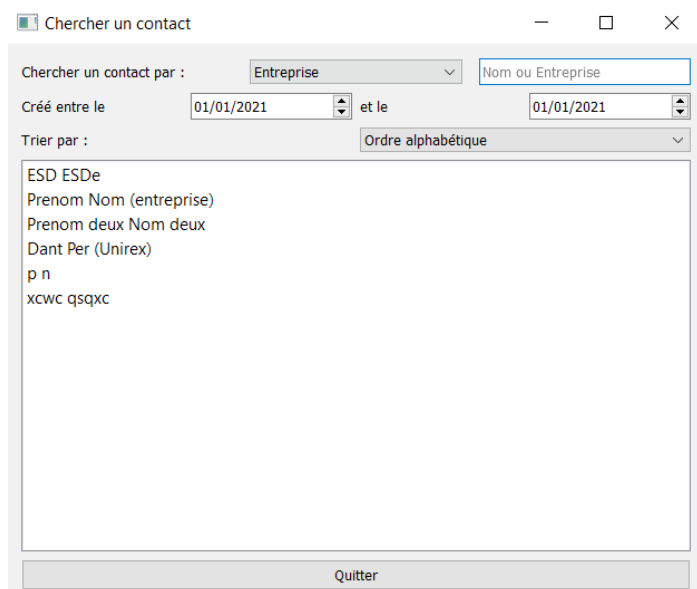
Créé entre le et le

Trier par : **Ordre alphabétique**

- Prenom Nom
- Prenom deux Nom deux

Quitter

Recherche d'un contact par son prénom ou son nom



Chercher un contact

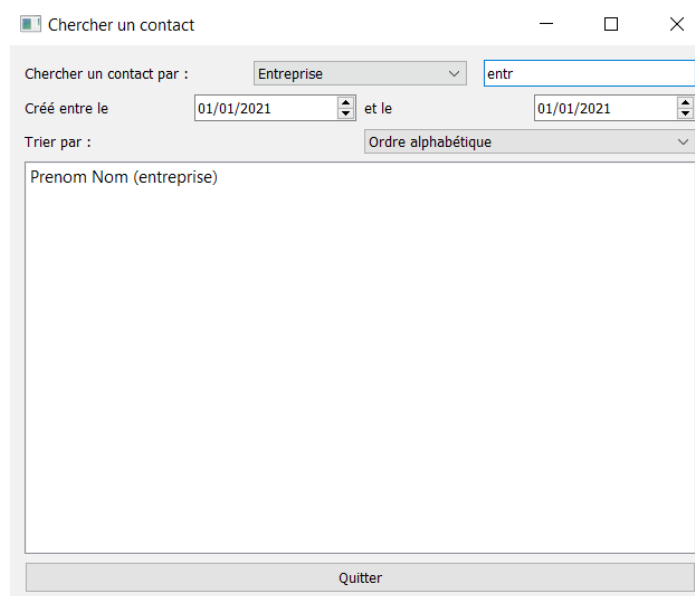
Chercher un contact par : **Entreprise**

Créé entre le et le

Trier par : **Ordre alphabétique**

- ESD ESDe
- Prenom Nom (entreprise)
- Prenom deux Nom deux
- Dant Per (Unirex)
- p n
- xcwc qsqxc

Quitter



Chercher un contact

Chercher un contact par : **Entreprise**

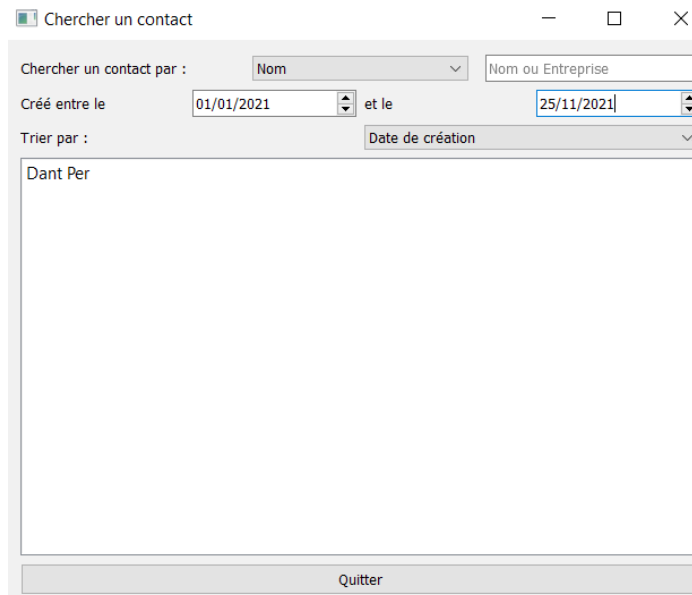
Créé entre le et le

Trier par : **Ordre alphabétique**

- Prenom Nom (entreprise)

Quitter

Recherche d'un contact par son entreprise



Recherche d'un contact dont la fiche a été créée entre deux dates

Comme nous pouvons le voir sur les images, le nom de l'entreprise des contacts est affiché (si le contact a une entreprise d'enregistrée dans sa fiche) lorsqu'on passe en recherche par entreprise. On peut également voir que le filtre par date de création de la fiche ne s'applique pas si les deux dates de l'intervalle sont mises au 01/01/2021.

Cette fenêtre et son fonctionnement sont gérés par la classe « RechercheContactWindow » dont le fonctionnement principal se base sur un simple algorithme de recherche et de tri dans une liste de contacts. En effet, pour afficher les résultats de la recherche effectuée par l'utilisateur, nous allons itérer parmi la liste de tous les contacts à disposition de la classe, pour tester si le nom/prénom de chaque contact contient le texte entré en recherche si nous sommes en mode de recherche par nom/prénom. Sinon, nous sommes en mode recherche par entreprise et le texte entré en recherche est alors testé dans l'entreprise de chaque contact. Ceci est réalisé avec la fonction `find` de la classe `string` et on détermine si la recherche est contenue en testant si `find` renvoie un position valide (donc non égale à `string::npos`).

Durant cette itération, nous pouvons également tester si la date de création se trouve entre les deux dates extrémités de l'intervalle de recherche. Chaque contact vérifiant tous ces tests (si applicables) sera ajouté à la liste des résultats, qui seront affichés dans une `QListView`. Cet affichage se fait par l'intermédiaire d'une `QStringList` à laquelle on va ajouter le `toString` de chaque contact résultat dans le cas d'une recherche par nom/prénom et sinon on va également concaténer l'entreprise (si le contact en a renseigné une) dans le cas d'une recherche par celle-ci. Il ne restera plus qu'à indiquer au modèle de la `QListView` la liste, avec `setStringList`.

4. Fenêtre d'affichage de fiche

Une fois un contact sélectionné, l'utilisateur peut ouvrir une fenêtre permettant d'afficher les informations dont le logiciel dispose sur le contact. En plus de les afficher, cette fenêtre permet également de modifier les informations (nom, prénom, entreprise, mail, téléphone, photo, ...), d'afficher les interactions du contact, d'en ajouter de nouvelles, de supprimer la fiche ou d'afficher des données supplémentaires comme la date de création de la fiche ou la date de dernière modification.

The screenshot shows a window titled 'Fiche du contact'. At the top is a profile picture of a pink bunny wearing a blue beret and a yellow bell. Below the photo are two buttons: 'Afficher les interactions' and 'Ajouter une interaction'. Underneath are input fields for 'Nom', 'Prénom', 'Entreprise', 'Téléphone', and 'Mail'. The 'Photo' field shows a file path and a 'Changer de photo' button. At the bottom, it displays 'Date de création de la fiche : 25/12/2021' and 'Date de dernière modification: 01/12/2021', followed by 'Valider les modifications', 'Supprimer le contact', and 'Quitter' buttons.

Visuel de la fenêtre d’affichage de la fiche d’un contact

Tous ces affichages sont gérés par la classe « FicheContactWindow », qui reprend exactement le même principe que la fenêtre de création de fiche pour l’affichage / la modification des informations et la sélection de la photo du contact (vérification de la cohérence, proposition d’une potentielle correction en cas d’erreur, ...). Lors de la demande de suppression de la fiche, un message de confirmation est demandé à l’utilisateur par l’intermédiaire de `QMessageBox::question` à l’image de la correction des entrées invalides.

L’affichage et l’ajout d’interactions est délégué à deux nouvelles fenêtres.

5. Affichage des interactions

L’affichage et la modification des interactions se fait dans une petite fenêtre à part, permettant de modifier le résumé d’une interaction ou sa date après l’avoir sélectionné dans la liste des interactions du contact courant.

The screenshot shows the 'Fiche du contact' window with a smaller 'Interactions de Prenom No...' window overlaid. The 'Interactions' window has a list of interactions: '24/11/2021 : Accident du travail @todo Passer un test d'aptitude à la médecine du tr...', '02/08/2021 : Retour de vacances @todo rattraper le travail @date 01/09/2021'. Below the list is a 'Détails :' section with a 'Date de l'interaction:' dropdown set to '02/08/2021' and a text area containing 'Retour de vacances @todo rattraper le travail @date 01/09/2021'. A 'Modifier le résumé' button is at the bottom of the details section. The background 'Fiche du contact' window shows the same contact information as before.

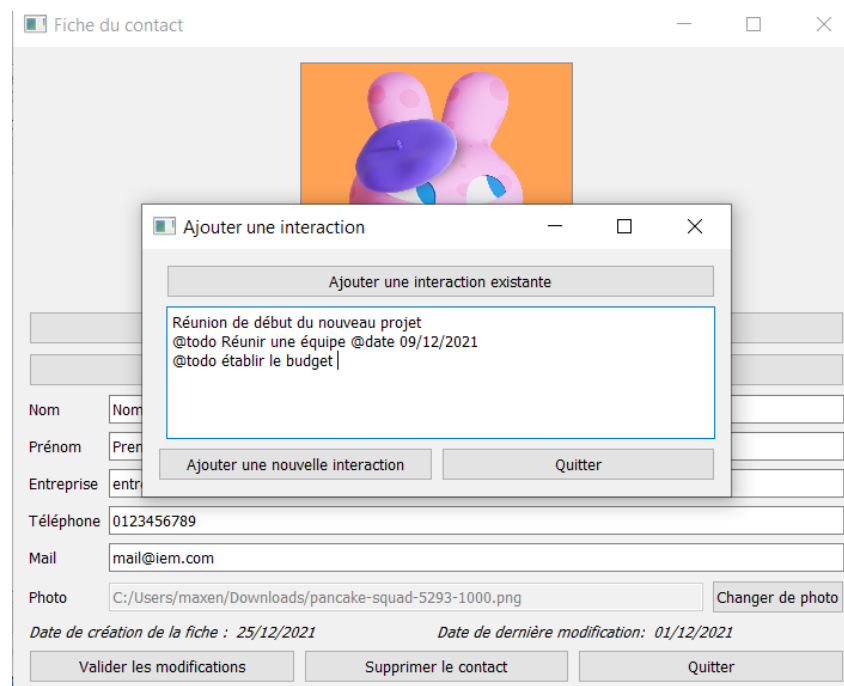
Visuel de l’affichage des interactions d’un contact

Ces fonctionnalités sont gérées par la classe «AfficheInteractionsWindow». Lors de la sélection d'un contact, elle va récupérer toutes ses interactions et les afficher par ordre décroissant de date, c'est à dire du plus récent au plus ancien. Ce tri est effectué grâce à la fonction sort des listes en utilisant le comparateur de dates « > ». Tout comme nous avons fait pour la liste des contacts, l'affichage se fait dans une QListView par l'intermédiaire de son modèle et d'une QStringList. Il est également possible de sélectionner une seule interaction dans la liste afficher pour obtenir des détails ou la modifier. Pour cela, nous allons écouter l'évènement de click sur un item de la liste des interactions et sélectionner, via la ligne de l'item qui correspond à l'indice, l'instance d'interaction correspondante dans une liste des interactions affichées que nous avons précédemment construite lors de l'affichage des interactions. Le même principe est utilisé pour sélectionner un contact dans la recherche mais ici des informations sont directement extraites et affichées sur la même fenêtre.

La zone de texte (QTextEdit), couplé au bouton pour modifier le résumé et le champ de date (QDateEdit) sont utilisables pour modifier l'interaction que l'utilisateur a sélectionné.

6. Ajout d'interactions

Afin d'afficher ou de modifier des interactions, il a bien fallu les ajouter. C'est le rôle de la fenêtre d'ajout des interactions permettant de taper le résumé d'une interaction (la date est mise automatiquement à la date du jour, si l'on note une interaction en retard il sera nécessaire de passer par la fenêtre des modifications afin de modifier la date).



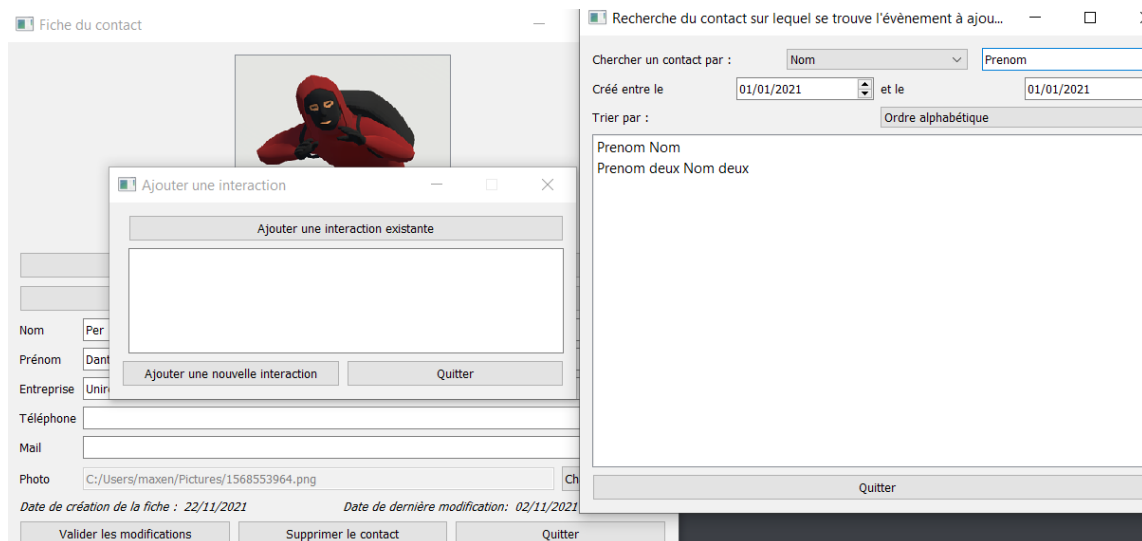
Visuel de la fenêtre d'ajout d'interactions

Cette tâche est confiée à la classe « AjoutInteractionWindow », qui va se contenter de récupérer le contenu de la zone de texte (QTextEdit) pour créer une nouvelle interaction et l'assigner au contact.

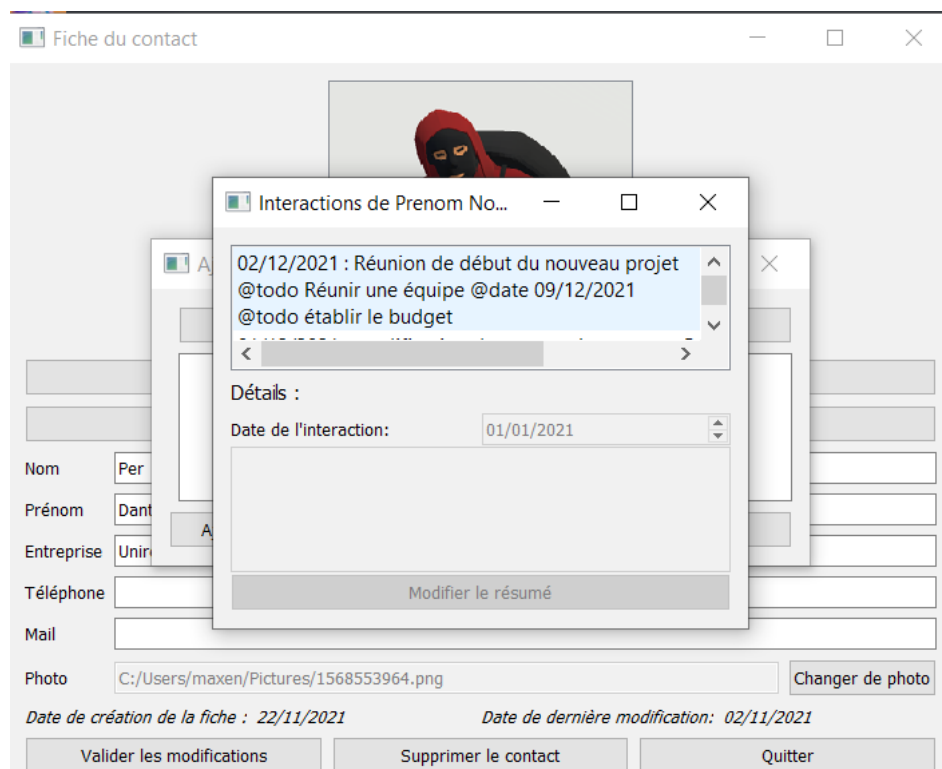
Il est également possible d'ajouter une interaction déjà existante sur un autre contact, dans le cas où plusieurs contacts différents auraient pris part à une même interaction. Il faudra alors utiliser le bouton « ajouter une interaction existante », ce qui déclenchera l'affichage d'une fenêtre de

recherche et sélection de contact, mais une autre instance que celle de la fenêtre principale tout en étant basée sur exactement la même classe RechercheContactWindow. Cette nouvelle recherche va permettre de sélectionner le contact sur lequel se trouve l'interaction à ajouter et la sélection d'un contact dans la fenêtre va déclencher l'affichage de la liste des interactions de l'autre contact (avec toujours une instance différente que celle de l'affichage de la fiche du contact, mais de la même classe). Pour finir, la sélection d'une interaction parmi celles affichées l'ajoutera au contact initial.

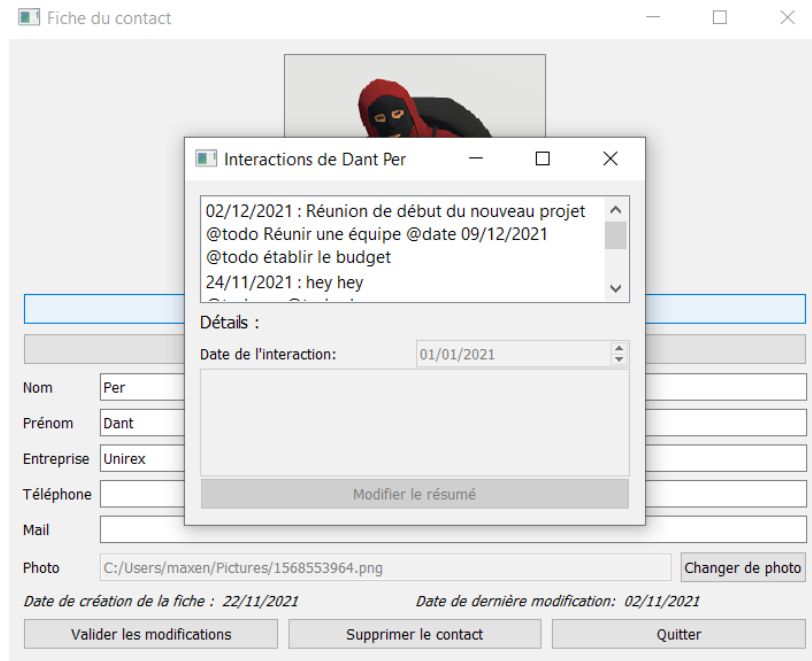
On souhaite ajouter au contact « Dant Per » l'interaction de la réunion ajoutée à « Prenom Nom » précédemment.



On va alors le rechercher dans la liste des contacts s'affichant après avoir cliqué sur « ajouter une interaction existante ».



Ses interactions s'affichent alors, on va sélectionner celle de la réunion.



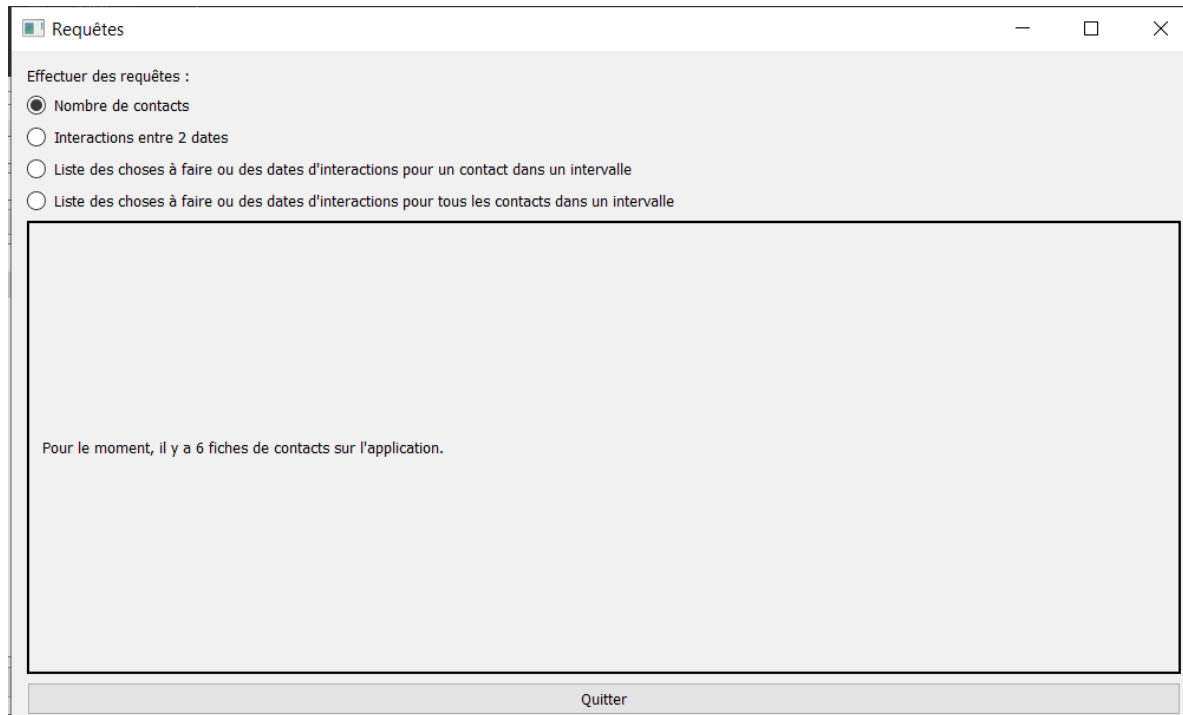
En affichant maintenant les interactions de « Dant Per », celle de la réunion est bien affichée.

Lorsque ceci est effectué, il s'agit bien de la même interaction pour tous les contacts auxquels elle a été ajoutée, sa modification via l'affichage des interactions d'un contact est donc « répercuté » (pas vraiment puisque c'est la même) sur tous les contacts impliqués. Les todos sont donc également en commun.

7. Requetes

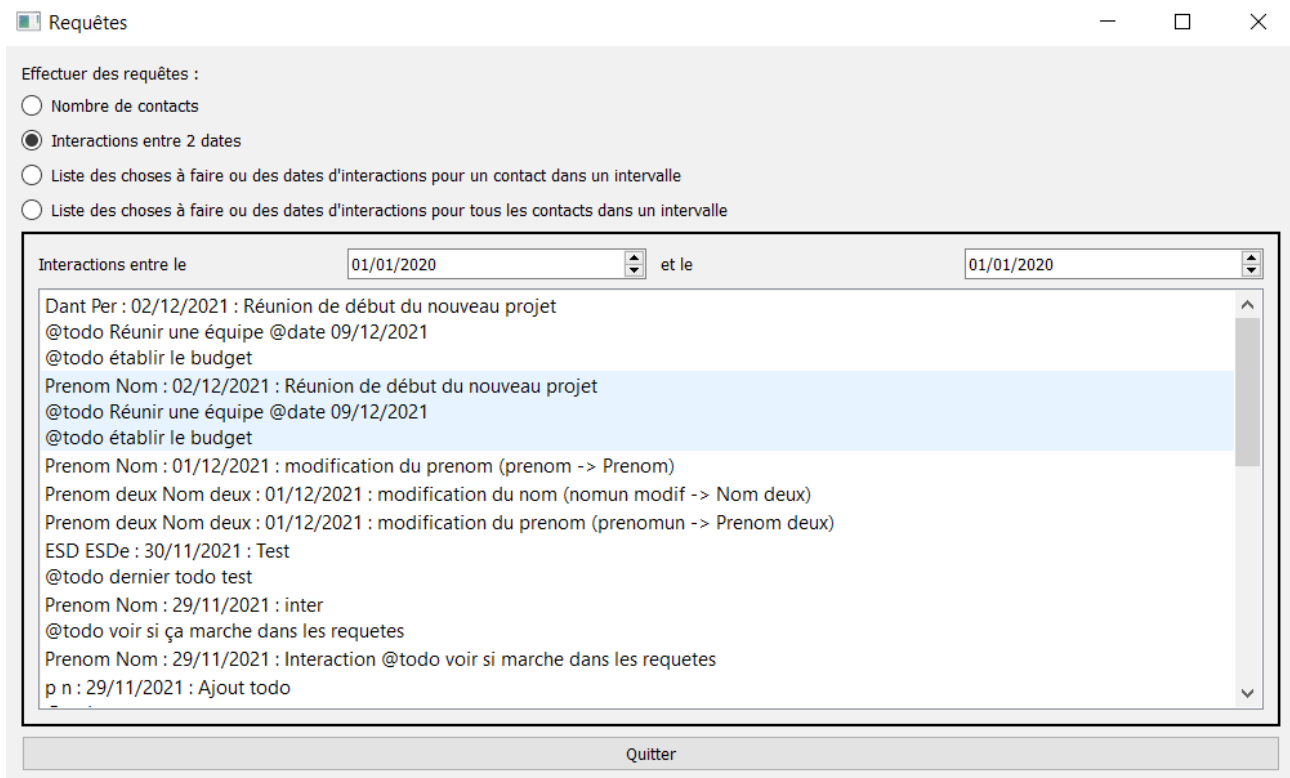
Pour mieux visualiser ces données, une fenêtre de requêtes est disponible depuis la fenêtre principale. Elle peut afficher le nombre de contacts, l'ensemble des interactions entre 2 dates, la liste des tâches à faire ou des dates d'interaction pour un contact (filtrage par intervalle de dates) ou la liste des tâches à faire ou des dates d'interaction pour l'ensemble des contacts (filtrage par intervalle de dates).

Chaque type de requête est sélectionnable via un bouton radio qui va déclencher le changement d'indice du widget affiché par le QStackedWidget central. La classe ayant la tâche de gérer toutes ces fonctionnalités est « RequeteWindow ».



L'affichage du nombre de contacts dans la fenêtre des requêtes

Pour afficher le nombre de contacts dans le logiciel, il suffit de compter le nombre d'éléments de la liste des contacts et de l'afficher dans un QLabel. On veillera également à modifier le texte lorsque la liste des contacts est mise à jour.



L'affichage de toutes les interactions, triées de la plus récente à la plus ancienne (toujours avec sort des listes).

Pour afficher toutes les interactions entre 2 dates, nous allons commencer par récupérer la liste de toutes les interactions à partir de tous les contacts. Cependant, comme nous l'avons évoqué lors de l'ajout des interactions il est possible que plusieurs contacts aient pris part à une même interaction. Nous allons donc composer une liste de paires Contact/Interaction en itérant parmis les tous les contacts pour ensuite les afficher avec le format « Contact : Interaction », dans une QListView comme évoqué plusieurs fois précédemment. Lors de l'ajout d'une paire, nous devons tester si la date de l'interaction est bien entre les dates d'extrémités entrées par l'utilisateur comme intervalle de recherche. Les interactions affichées sont également triées par date de la plus récente à la plus ancienne. Si les deux dates sont mises au « 01/01/2021 », on va afficher toutes les interactions par convention.

Requêtes

Effectuer des requêtes :

☐ Nombre de contacts

☐ Interactions entre 2 dates

☒ Liste des choses à faire ou des dates d'interactions pour un contact dans un intervalle

☐ Liste des choses à faire ou des dates d'interactions pour tous les contacts dans un intervalle

Dates entre le et le

Afficher :

☐ Seulement non effectués

- ☐ TODO pour le 01/09/2021 : @todo rattraper le travail @date 01/09/2021
- ☐ TODO pour le 25/11/2021 : @todo Passer un test d'aptitude à la médecine du travail @date 25/11/2021
- ☒ TODO pour le 02/12/2021 : @todo établir le budget
- ☐ TODO pour le 09/12/2021 : @todo Réunir une équipe @date 09/12/2021
- ☒ TODO pour le 01/02/2023 : @todo Todo date @date 01/02/2023

Quitter

La liste des choses à faire pour le contact sélectionné (« Prenom Nom »)

Pour afficher la liste des choses à faire pour un contact donné (celui sélectionné pour tout le logiciel par la recherche depuis la fenêtre principale), on va récupérer sa liste de todos et tester si chacun vérifie les conditions pour être affiché : date dans l'intervalle (ou les dates sont au 01/01/2021) ou seulement les non effectués. S'il passe les tests nécessaires, on va l'afficher avec un QStandardItem et non plus une QStringList afin de rendre l'item checkable. Son état (checké ou non) est déterminé selon si le todo a été effectué ou non. On veillera également à mettre en texte de l'item le contenu du todo. Pour les ajouter dans la QListView, on va cette fois passer par un QStandardItemModel, auquel on va ajouter chaque item via appendRow. Si le todo est non effectué et que sa date d'échéance est dépassée, on pourra afficher son item en rouge pour indiquer visuellement à l'utilisateur les tâches les plus importantes. Pour finir, on va lister tous les todos affichés pour que lors du check ou decheck d'un item dans la liste affichée, on puisse faire la correspondance et mettre effectué ou non sur l'instance de todo correspondante.

Dans le cas où on veut afficher les dates, on va récupérer toutes les interactions du contact et afficher chaque date d'interaction (si elle est dans l'intervalle) dans une QListView avec une liste de string comme précédemment.

Requêtes

Effectuer des requêtes :

☐ Nombre de contacts

☐ Interactions entre 2 dates

☐ Liste des choses à faire ou des dates d'interactions pour un contact dans un intervalle

☒ Liste des choses à faire ou des dates d'interactions pour tous les contacts dans un intervalle

Dates entre le 01/01/2020 et le 01/01/2020

Afficher : la liste des todos pour les contacts

☒ Seulement non effectués

- ☐ Dant Per : TODO pour le 01/09/2021 : @todo rattraper le travail @date 01/09/2021
- ☐ Prenom Nom : TODO pour le 01/09/2021 : @todo rattraper le travail @date 01/09/2021
- ☐ Dant Per : TODO pour le 01/11/2021 : Deuxieme todo modif
- ☐ Dant Per : TODO pour le 25/11/2021 : @todo deux
- ☐ Prenom Nom : TODO pour le 25/11/2021 : @todo Passer un test d'aptitude à la médecine du travail @date 25/11/2021
- ☐ ESD ESDe : TODO pour le 29/11/2021 : @todo besoin de todos 1
- ☐ Dant Per : TODO pour le 09/12/2021 : @todo Réunion une équipe @date 09/12/2021
- ☐ Prenom Nom : TODO pour le 09/12/2021 : @todo Réunion une équipe @date 09/12/2021

Quitter

*La liste des choses à faire pour l'ensemble des contacts,
en filtrant seulement les tâches non effectuées*

Cette fois, nous voulons faire la même chose que pour le contact sélectionné mais avec l'ensemble des contacts. On va donc itérer parmi tous les contacts et ajouter dans une liste une paire de texte + Todo pour chaque Todo de chaque contact dont les tests sont vérifiés (date dans l'intervalle, non effectué si seulement les non effectués doivent être affichés, ...). Le texte contient la concaténation du contact et du contenu du todo. L'intérêt de cette liste de paires est de pouvoir la trier selon les dates d'échéance du todo, de la plus proche (ou dans le passé) à la plus lointaine. On va ensuite pouvoir afficher des items checkables dans la liste comme pour le cas à un seul contact.

Pour afficher la liste des dates des contacts, on va utiliser une map de Date sur une liste de contacts. On va donc itérer parmi la liste de tous les contacts, pour récupérer pour chaque contact toutes ses interactions et ajouter dans la map avec pour clé la date de l'interaction le contact en question. On va ensuite itérer parmi les clés de la map (donc toutes les dates où des contacts ont interagi) pour supprimer les doublons parmi la liste des contacts à cette date (il est inutile de les afficher plusieurs fois) et on va pouvoir ajouter dans la QListView un item textuel de la forme « Date : Contact1, Contact2, ..., ContactN » avec les contacts dans la liste des contacts ayant interagi à cette date.

Pour finir, nous pouvons résumer les affichages des fenêtres avec le schéma suivant.

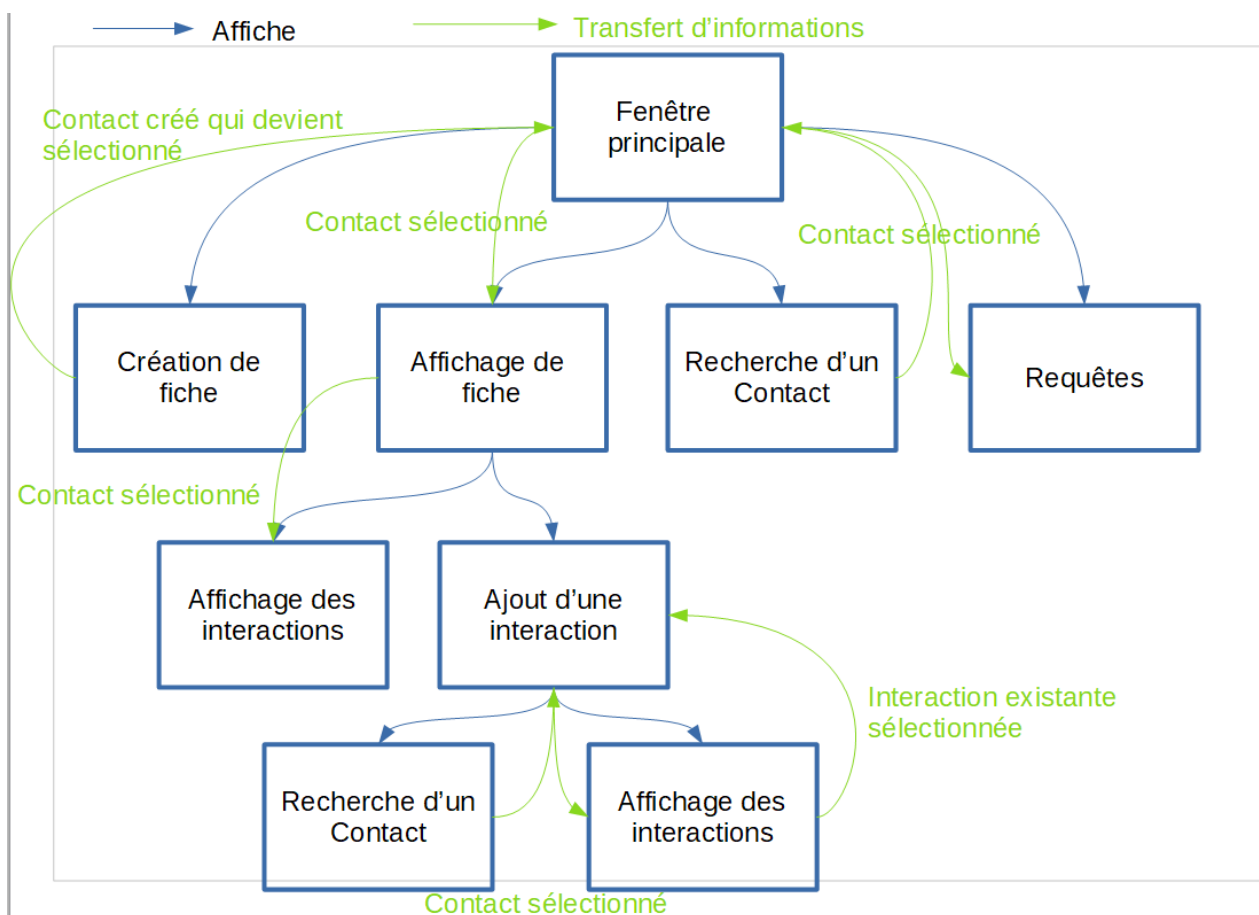


Schéma récapitulatif de l'affichage des fenêtres et des liens entre elles

II. Gestion interne

Nous avons vu comment l'utilisateur pouvait interagir avec notre logiciel pour créer ou modifier des informations et nous allons voir maintenant comment ces informations sont gérées du côté du logiciel. Cette partie a été pratiquement entièrement réalisée pour la jalon 1 du projet et il s'agit de la partie programmation orientée objet.

La gestion interne se base sur une classe « intelligente » GestionContact et des classes de données telles que Contact, Interaction ou Todo. Nous utilisons également une classe Date faisant office d'interface pour stocker et utiliser les dates. Nous aurions pu utiliser des classes d'associations pour stocker les liens entre contact et interaction, ou interaction et todo, mais nous avons préféré référencer directement par des listes de pointeurs les interactions d'un contact et les todos d'une interaction.

Les listes contiennent des pointeurs, ce qui est utile pour laisser la possibilité à plusieurs contacts de pointer sur une unique interaction, dans le cas où plusieurs contacts auraient pris part à une même interaction (exemple : Plusieurs clients prennent part à la même réunion).

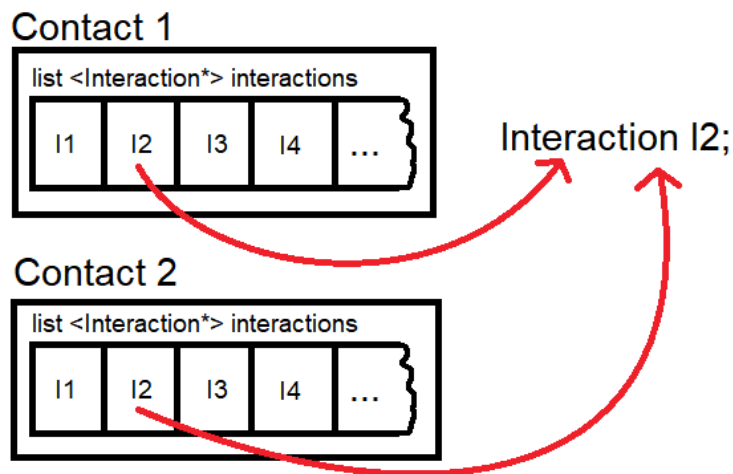


Schéma de plusieurs contacts ayant une même interaction

Cette fonctionnalité supplémentaire peut certes être pratique mais apporte un problème supplémentaire : il faut faire attention quand on veut supprimer en mémoire les contacts. En effet, le destructeur de Contact se charge de détruire toutes les instances d'interaction pointées par la liste d'interactions du contact. Or, une fois l'exécution terminée on détruit l'instance de GestionContact (et donc par extension toute la liste des contacts). Mais dans le cas où plusieurs contacts pointent vers la même instance d'interaction, on pourrait tenter de supprimer une interaction déjà supprimée.

Pour détruire correctement les données en mémoire, le destructeur de la classe GestionContact est un peu particulier : au lieu de supprimer aveuglement par rapport à l'ensemble des contacts les interactions dans le destructeur de Contact, il nous est utile de remonter au destructeur de la classe GestionContact car il permet d'avoir une vue d'ensemble de toutes les interactions de tous les contacts et il va pouvoir récupérer toutes les interactions de tous les contacts, supprimer tous les doublons et détruire toutes les instances d'interaction, puis tous les contacts de la liste.

Il est possible d'accéder indirectement aux todos d'un contact par le biais de ses interactions (qui pointent vers leurs todos). Pour plus de facilité, nous avons donc ajouté un accesseur virtuel à Contact qui sort la liste de tous les todos de l'instance de Contact. Pour cela, nous avons juste à récupérer et à indiquer dans une liste les todos de chaque interaction.

Pour finir, nous pouvons résumer notre gestion interne des différents objets avec un diagramme des classes.

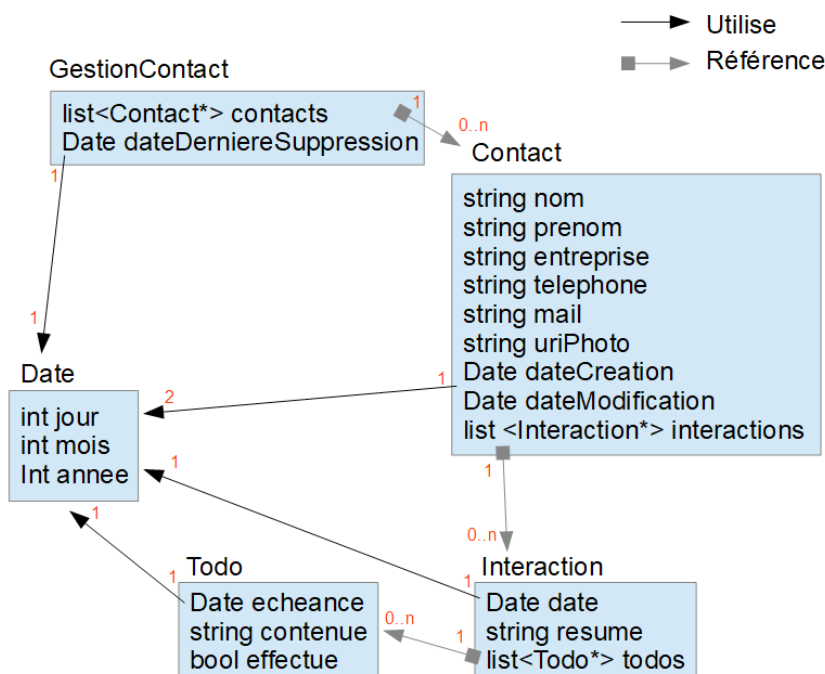


Diagramme des classes

III. Base de données

Lorsque l'utilisateur note soigneusement les informations d'un contact et ses interactions ou note qu'une tâche a été effectuée, il aimerait bien que ces données soient précieusement sauvegarder pour que le lendemain, le logiciel soit toujours capable de se rappeler ce qui a été noté la veille. Pour cela, nous stockons les informations des différents objets manipulés par notre logiciel dans une base de données SQLite. QT nous permet une connexion facile avec la base de données grâce au module QSql/QSqlQuery. Il suffit d'indiquer le driver à utiliser avec `QSqlDatabase::addDatabase("SQLITE")` puis d'indiquer le chemin vers la base de données (le fichier donc puisque nous utilisons SQLite) avec `setDatabaseName` sur l'objet renvoyé lors de l'ajout de la base de donnée. Il ne nous reste plus qu'à exécuter des requêtes en manipulant des objets de la classe `QSqlQuery` avec du code SQL. Nous allons soit utiliser directement `QSqlQuery.exec(requete)` pour des requêtes uniquement de simple consultation (`SELECT ...`) ou nous allons préparer la requête avec des paramètres via `QSqlQuery.prepare(requete paramétrée)` où les paramètres seront de la forme « :param ». On pourra alors assigner une valeur à un paramètre avec la méthode `QSqlQuery.bindValue(«:param », val)`, avec val la valeur d'un attribut d'une instance par exemple.

Par exemple pour rechercher les contacts ayant comme nom « Dupont », nous exécuterions les procédures suivantes : `QSqlQuery query ; query.prepare(« SELECT * FROM contacts WHERE nom = :nom ; ») ; query.bindValue(«:nom », « Dupont ») ; query.exec() ;` Bien sûr dans une réelle application, « Dupont » aurait été remplacé par l'attribut nom de contact afin d'obtenir des requêtes paramétrables selon les instances utilisées.

Afin de stocker les valeurs de chaque type d'objet utilisé par le logiciel, nous avons décidé de créer une table par type d'objet. Nous avons donc une table Contact, une table Interaction et une table Todo. Chacune de ces tables va posséder un attribut pour chaque attribut de l'objet : la table Contact contient un attribut nom, un prénom, un téléphone, ... la table Interaction contient un attribut résumé et une date, un todo contient un contenu et une date d'échéance.

De plus chacune de ces tables va contenir un entier Id comme clé primaire pour identifier de manière unique chaque objet et cela nous évite par exemple le problème du cas où plusieurs contacts ont le même nom et prénom.

Cet identifiant nous permet également de faire des liens entre les objets : la table Todo contient également un attribut « InteractionId » qui est une clé étrangère pointant vers l'identifiant de la table des interactions. De cette façon, on peut retrouver à quelle interaction appartient chaque Todo et nous sommes sûr qu'elle est unique (et existante) grâce au fait que ce soit une clé étrangère. Nous aurions pu utiliser le même principe pour lier les interactions aux contacts (une clé étrangère ContactId dans la table Interaction). Cependant, nous avons laissé la possibilité que différents contacts aient pris part à une même interaction et nous avons donc créé une table d'association ContactsToInteractions. Cette table contient deux attributs : ContactId et InteractionId, des clés étrangères pointant vers les identifiants de Contact et d'Interaction. Une ligne dans cette table indique donc l'association entre un contact et une interaction, laissant le champ libre pour plusieurs contacts pointant vers une interaction.

Toutes ces clés étrangères nous sont également très utiles pour les suppressions grâce au mécanisme des cascades. En effet, en paramétrant les clés étrangères avec l'option cascade, la suppression de la ligne vers laquelle pointe une clé étrangère entraîne également la suppression de la ligne pointant vers celle-ci. Par exemple, la suppression d'un contact entraîne la suppression des associations Contact-Interaction ou la suppression d'une interaction entraîne automatiquement la suppression de tous ses todos dans la base de données.

Nous avons également une table Infos pour stocker la date de la dernière suppression.

Nous pouvons résumer le modèle de notre base de données avec le schéma suivant.

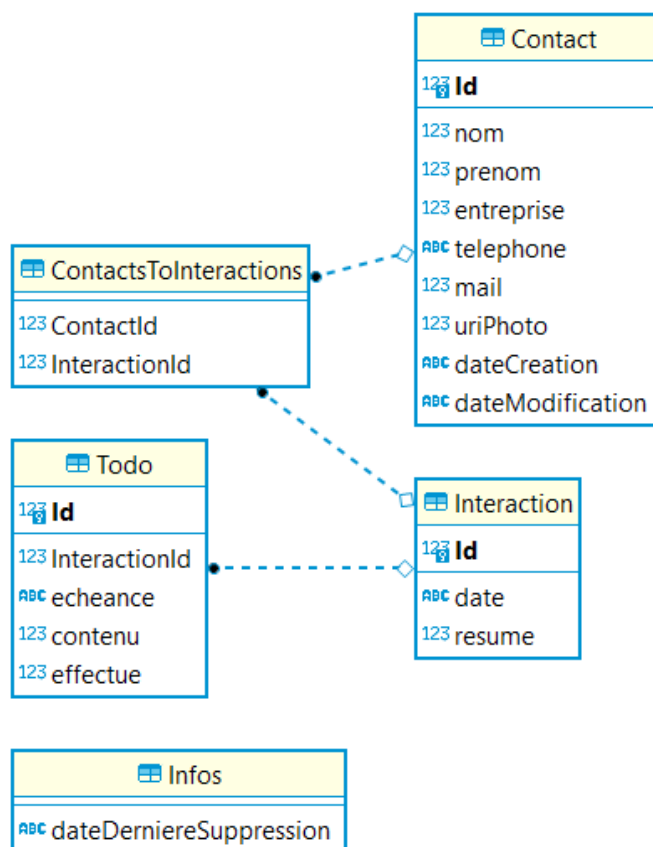


Schéma du modèle de la base de données (généré via dbeaver)

Tout ceci est géré dans le logiciel par une classe « DatabaseManager ». Cette classe va initier la connexion avec la base de données comme expliqué au début de cette partie et lancera toutes les requêtes nécessaires lors de la manipulation des données. Lors de son initialisation, elle va également recréer la structure de la base de données, dans le cas où le fichier ait été supprimé entre deux exécutions ou n'ait jamais existé. Pour cela, nous allons ouvrir un fichier SQL modèle et exécuter toutes les lignes s'y trouvant (en ignorant les commentaires). Le fichier modèle aura préalablement été généré en exportant la structure de la base de données. Sa lecture est réalisée grâce à un QFile et un QTextStream pour faciliter la lecture en utilisant readLine. On va également tester si des informations sont présentes dans la table Infos et si non, on va insérer la valeur par défaut « 01/01/2021 » comme date de dernière suppression.

Ensuite, il va falloir charger dans le logiciel toutes les données préalablement sauvegardées dans la base de données. Nous allons commencer par lire et créer les contacts. Pour cela, on va exécuter la requête « SELECT * FROM Contact; » et on va lire chaque ligne du résultat en utilisant QSqlQuery.next() dans une boucle while (next renverra faux lorsque nous arriverons à la fin du résultat). Lorsqu'on lit une ligne, on peut extraire les valeurs de chaque attribut en utilisant QSqlQuery.value(indice) avec indice l'indice de l'attribut dans la table. Il ne nous reste plus qu'à transformer le QVariant résultat dans le type désiré. Nous utilisons une map pour faire une correspondance entre l'identifiant des contacts dans la base de données et les instances de Contact dans le logiciel, ce qui nous permet d'éviter les recherches pour manipuler les contacts.

Il nous faut ensuite charger les interactions, ce que nous faisons avec un « `SELECT * FROM Interaction;` ». Ceci nous permet de récupérer les données de chaque interaction mais il nous faut également les lier aux contacts correspondants, ce que nous faisons en exécutant une autre requête, cette fois sur la table des liens `ContactsToInteractions` : « `SELECT ContactId FROM ContactsToInteractions WHERE InteractionId = :id;` » où on va appeler `bindValue` avec l'identifiant de l'interaction que nous sommes en train de charger. On va alors pouvoir créer l'interaction avec le premier contact trouvé (dont on retrouve l'instance très rapidement grâce à son identifiant que l'on passe dans la map `identifiant → instance`) et l'ajouter pour tout contact supplémentaire avec laquelle elle a un lien. Sur le même principe que pour les contacts, nous utilisons une map `identifiant → instance` pour les Interactions.

Une fois qu'une interaction a été chargée, nous devons également lui rappeler ses todos, ce qui est fait avec une requête : « `SELECT * FROM Todo WHERE InteractionId = :id;` » où `id` sera `li` (`bindValue`) avec l'identifiant de l'interaction qui vient d'être chargée. On peut ensuite lui ajouter chacun des todos grâce aux lignes résultats de la requête.

Nous aurions très bien pu nous passer du stockage des tâches dans la base de données car nous pouvons les retrouver en analysant à nouveau le résumé de chaque interaction. Cependant, nous avons décidé d'ajouter l'attribut « effectué ou non » sur les todos, nous permettant de sauvegarder si une tâche a été réalisée ou non et cela nous oblige à stocker son état dans la base de données et donc tout le todo.

Une fois que les contacts, les interactions et leurs todos ont été lus et chargés dans l'application, elle peut démarrer et utiliser ces objets. Durant leur manipulation, ils peuvent être amenés à être modifiés ou à ce que de nouveaux objets soient créés, ce qui nécessite d'être enregistré dans la base de données.

Lors de la création (par `GestionContact`) d'un nouveau contact, nous devons également créer la ligne correspondante dans la base de données. Le `DatabaseManager` va donc exécuter une requête d'insertion : « `INSERT INTO Contact (nom, prenom, entreprise, telephone, mail, uriPhoto, dateCreation, dateModification) VALUES (:nom, :prenom, :entreprise, :tel, :mail, :photo, :dateCreation, :dateModification);` » et chaque paramètre sera lié avec l'attribut du nouveau contact correspondant. Nous procéderons de la même façon lors de la création d'une Interaction ou d'un Todo. On veillera également à récupérer l'identifiant de l'objet dans la base de données pour l'ajouter dans la map appropriée pour lier les instances à leur sauvegarde dans la base de données.

Lors de la création d'une Interaction ou de l'ajout d'une interaction à un contact supplémentaire, nous veillerons bien à également insérer dans la table des liens entre contacts et interactions, `ContactsToInteractions`, une ligne avec l'identifiant du contact et l'identifiant de l'interaction. Cette insertion sera faite après la création de l'interaction afin de respecter la contrainte de clé étrangère. Les identifiants seront retrouvés facilement grâce aux maps utilisées.

De même, lorsqu'un des attributs est modifié dans l'application, il faut répercuter ce changement dans la base de données afin de le rendre persistant. Le `DatabaseManager` va cette fois exécuter une requête de modification, du type : « `UPDATE Contact SET attribut = :nouvelle_valeur WHERE Id = :id;` » avec les `bindValue` correspondants (et attribut sera l'attribut modifié en question) et l'identifiant retrouvé grâce à la map. Nous procédons de la même façon pour les modifications des interactions. Cependant, les todos seront régénérés en cas de modification du résumé d'une interaction. Les anciens seront donc supprimés. Le statut `fais ou non` d'un todo est également modifié avec un `update`.

Il peut arriver que la fiche d'un contact soit supprimé. Le `DatabaseManager` devra alors utiliser un dernier type de requête : « `DELETE FROM Contact WHERE Id = :id;` ». Comme nous l'avons précisé au début de cette partie, il n'est pas nécessaire de supprimer ses interactions (et leur

lien) puisque les cascades des clés étrangères dans la structure de la base de donnée le réalise toutes seules, ce qui nous permet d'alléger la gestion du côté de l'application.

IV. Connexion entre parties

Dans les précédentes parties, nous avons expliqué comment chaque composant fonctionne de manière individuelle. Cependant, ces composants doivent former un tout pour créer l'application et communiquer entre eux dès qu'un événement se produit. Nous réalisons ceci grâce au principe des Signaux/Slots de QT. Un signal représente un « événement » que nous allons pouvoir connecter à un slot. Un signal est déclenché avec le mot clé `emit signal(paramètres)` et nous pouvons connecter grâce à la fonction `connect(émetteur, SIGNAL(signal(...)), receveur, SLOT(slot(...)))`. Tous nos connect sont effectués dans les constructeurs des classes concernées.

1. Communication entre fenêtres

Comme nous l'avons vu lors de la partie sur l'interface graphique, les fenêtres se communiquent entre elles des informations, ce qui est réalisé grâce à des signaux et des slots.

Lors de la recherche des contacts, la sélection d'un élément de la liste déclenche un signal `contactSelected`, qui aura été connecté préalablement au slot `updateContactValues` de la fenêtre principale. Le slot va ensuite se charger de modifier l'affichage de la fenêtre principale pour indiquer quel contact a été sélectionné. L'événement aura également été connecté à un slot de la fenêtre d'affichage de fiche (pour afficher les informations du contact sélectionné) et au slot de la fenêtre des requêtes (pour que ce soit ses interactions et todo qui soient affichés).

De la même façon, la fenêtre d'affichage de la fiche d'un contact communique avec ses sous fenêtres d'affichage des interactions du contact et d'ajout d'interaction. Lorsque son slot de sélection d'un contact est appelé par le signal de la fenêtre principale, il est émis à nouveau par la fenêtre d'affichage de la fiche car cet autre signal est connecté aux slots de sélection de ses 2 sous fenêtres, afin qu'elles affichent les interactions du contact et ajoute l'interaction au bon contact.

Pour finir, la fenêtre d'ajout des interactions communique avec ses sous fenêtres lors de l'ajout d'une interaction déjà existante sur un autre contact. En effet, elle fait appel à une fenêtre de recherche pour rechercher le contact sur lequel se trouve l'interaction voulue. Lorsqu'un contact est sélectionné dans cette nouvelle fenêtre de recherche, le signal de sélection d'un contact sera déclenché mais sera cette fois connecté à une fenêtre d'affichage des interactions d'un contact. La sélection d'une interaction sur cette nouvelle fenêtre d'affichage des interactions sera connectée au signal d'ajout d'une interaction au contact courant de la fenêtre initiale d'ajout d'une interaction.

Toutes ces connexions sont résumées dans le schéma des fenêtres de la première partie.

2. Communication générale

Il nous faut également établir des connexions entre les fenêtres, la gestion interne de l'application et le gestionnaire de la base de données. Pour cela, nous devons utiliser un intermédiaire, `GestionnaireQObject`, car la gestion interne (`GestionContact`) n'est pas un objet de QT mais des classes indépendantes. `GestionnaireQObject` est donc au centre de ces différentes parties et a pour rôle de les connecter entre elles. Il va connecter les signaux envoyés par les

fenêtres (représentées par la fenêtre principale) à ses slots qui appelleront les méthodes de GestionContact ou à des slots du DatabaseManager.

Ce gestionnaire va également renvoyer à toutes les fenêtres (via la MainWindow) des résultats ou des changements effectués par GestionContact (ajout d'un contact à la liste). C'est également lui qui va transmettre à GestionContact le chargement des instances stockées dans la base de données via des signaux et slots appropriés.

C'est également la que la transformation des données en JSON ou la lecture depuis JSON est effectuée.

Nous pouvons résumer toutes ces connexions et communications avec le schéma suivant.

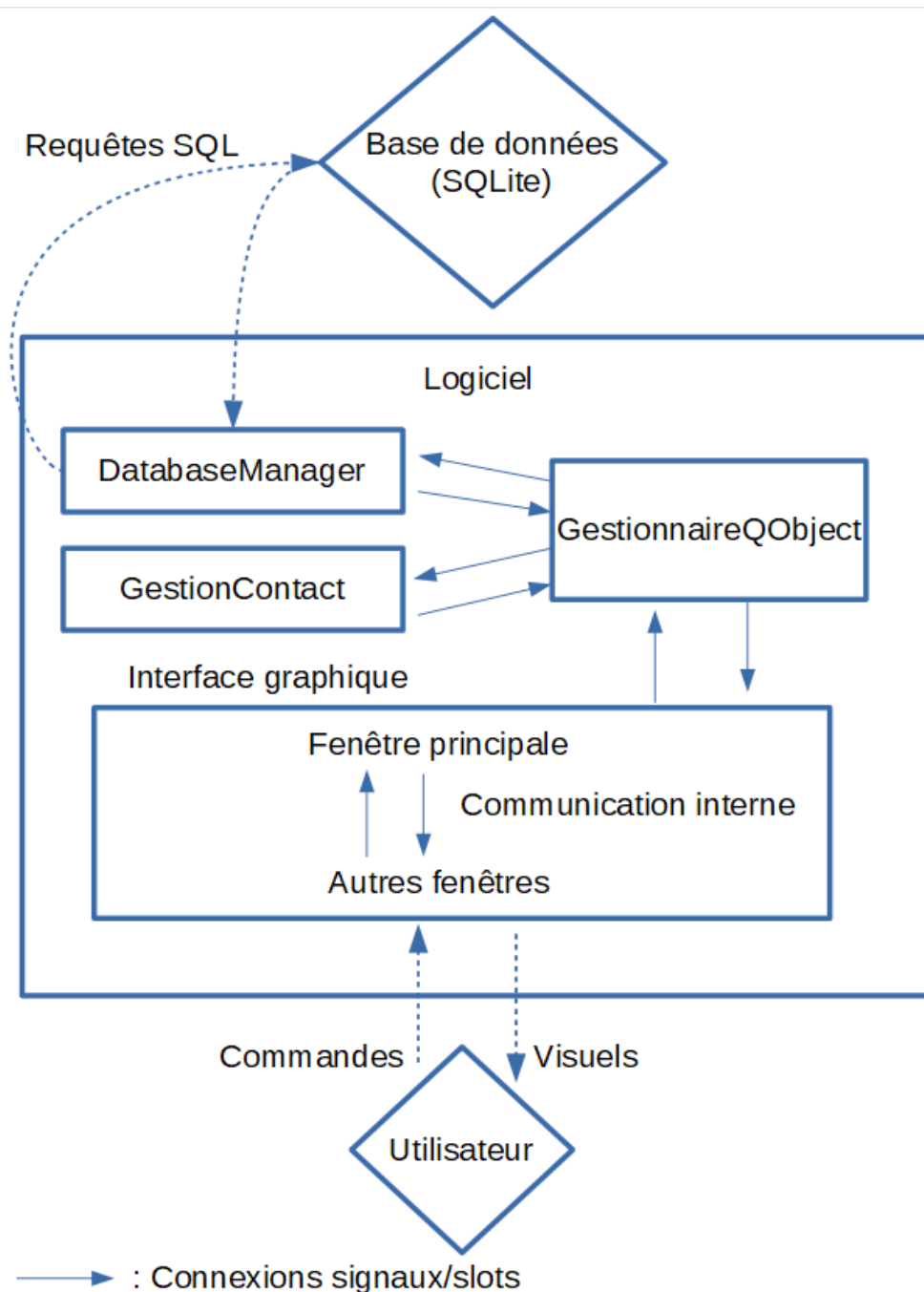


Schéma des différentes connexions entre les parties ou éléments externes

Conclusion : Nous avons construit un logiciel de gestion de relations clients se basant sur 3 parties (interface graphique, gestion interne, base de données) et communiquant entre elles grâce au système de signaux/slots de QT. Nous aurions également pu ajouter quelques fonctionnalités : l'ajout automatique et configurable d'interactions lors d'évènements (par exemple ajouter un todo pour souhaiter la bienvenue à un contact lorsque sa fiche a été créée) ou l'automatisation de certaines tâches (envoi de mail programmé pour souhaiter la bienvenue par exemple).