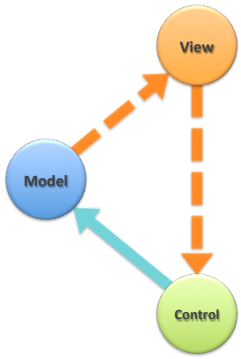
ensiie
école nationale supérieure d'informatique pour l'industrie et l'entreprise

Projet LOA

On souhaite réaliser une application QT en utilisant l'architecture MVC.

1. L'architecture MVC dans QT



L'architecture MVC (Model / View / Controller) impose la séparation entre les données (le modèle), la présentation de ces données (la vue) et l'édition de ces données entre la vue et le modèle (le contrôleur).
Dans la plupart des interfaces utilisateurs (comme dans Swing en Java par exemple) le contrôleur est intégré à la vue. C'est aussi le cas en QT, mais celui-ci vous permet (au besoin) de personnaliser le contrôleur d'une vue au travers d'un délégué qui permet :

- de personnaliser l'édition des éléments au moyen d'un éditeur ;
- de personnaliser le rendu des éléments à l'intérieur d'une vue.

QT fournit un certain nombre de classes pour vous aider à implémenter cette architecture (voir Figure 1 ci-dessous) :

- Tous les modèles fournis par QT sont basés sur la classe *QAbstractItemModel* et ses héritières (abstraites ou concrètes) qui fournit une interface pour accéder aux données du modèle en utilisant des indexs (QModelIndex) utilisables par l'ensemble des vues standard QT.
- Toutes les vues associées à un modèle fournies par QT sont basées sur la classe *QAbstractItemView* et ses héritières telles que QTableView, QListView ou QTreeView qui fournissent des vues/contrôleurs standards qui permettent d'afficher (au moins) la structure de votre modèle.
- Tous les délégués QT sont basés sur la classe *QAbstractItemDelegate* et la plupart des vues standard utilisent un QStyledItemDelegate héritière de *QAbstractItemDelegate* comme délégué par défaut.

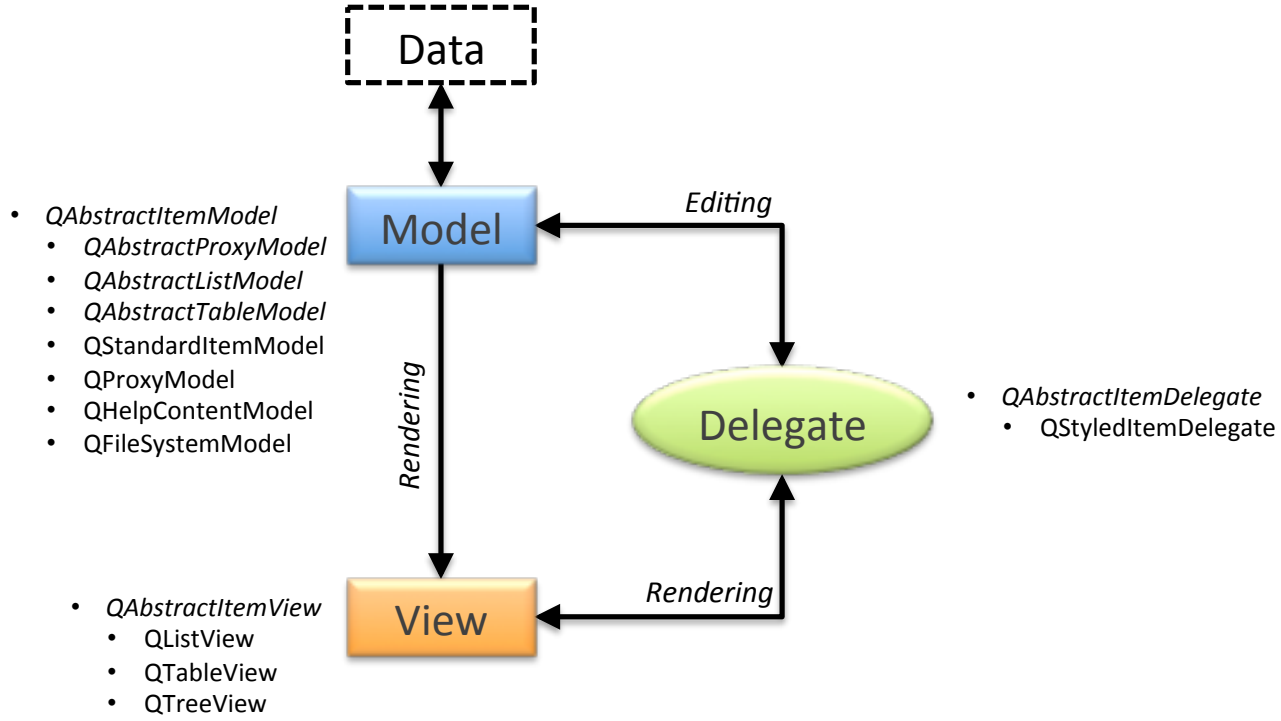


Figure 1 : l'architecture MVC dans QT

2. Modèle et opérations

Le « modèle » de données de votre application contient les données manipulées par votre application et doit se conformer à une des interfaces de *QAbstratItemModel* ou une de ses descendantes pour être affiché correctement dans une vue standard de Qt.
Ce modèle devra contenir une collection d'éléments éventuellement eux même constitués de sous éléments.
Le programme doit pouvoir gérer la collection d'éléments de votre modèle, il doit donc pouvoir :

- Afficher les éléments de votre modèle dans la partie gauche de l'UI.
- Afficher les détails de l'élément sélectionné dans la partie gauche de l'UI dans la partie droite.
- Éditer le contenu de l'élément sélectionné dans la partie gauche de l'UI dans la partie droite.
- Créer un nouvel élément.
- Détruire un élément.
- Rechercher le ou les éléments satisfaisant un ou plusieurs critères (filtrage).
- Enregistrer la liste des éléments dans un fichier en utilisant le format de votre choix (XML ou JSON par exemple).
- Lire (ou ajouter) une liste d'éléments depuis un fichier.

3. Structure de l'interface graphique

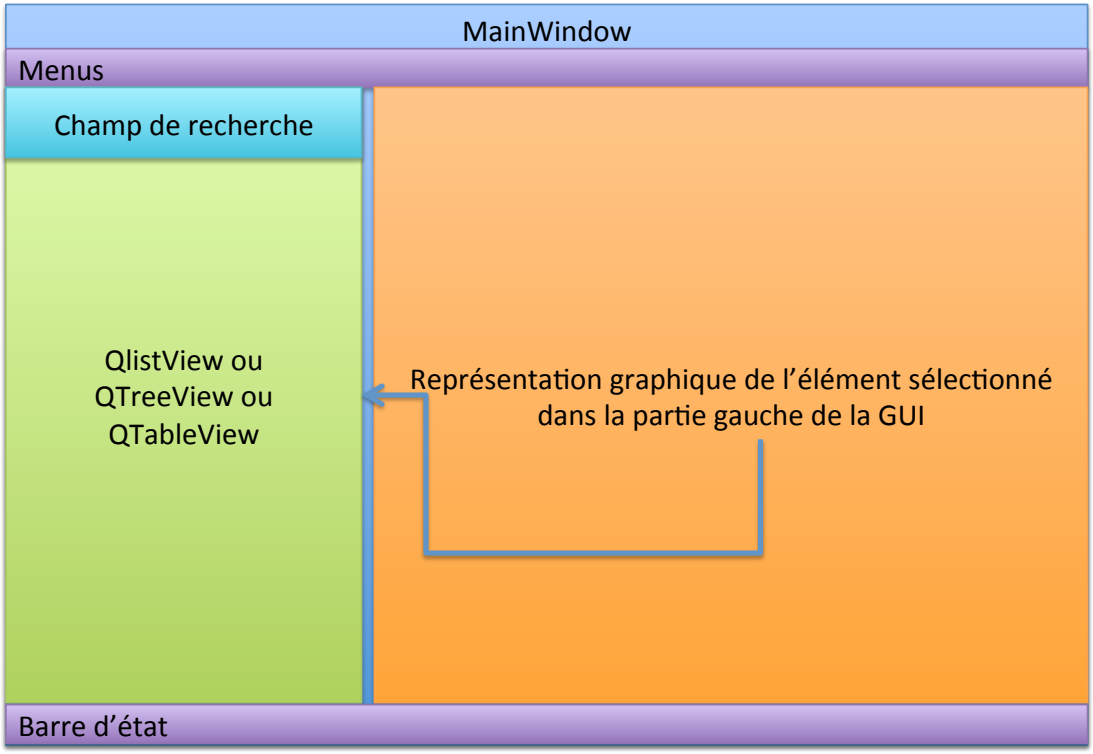


Figure 2 : Structure de l'interface graphique

L'interface graphique devra contenir les éléments ci-dessus (mais pas forcément avec la même disposition).
La partie droite du widget central devra contenir l'affichage (et/ou l'édition) d'un élément sélectionné dans la partie gauche qui utilise une vue standard.
Le champ de recherche pourra être utilisé pour rechercher (et afficher dans la partie gauche) un ou plusieurs éléments correspondant à un même critère de recherche.

4. Travail demandé

Ce projet est à faire en monôme ou en binôme en utilisant l'API QT.

Implémentez un programme de gestion d'éléments en respectant l'architecture MVC (Modèle / Vue(s) / Contrôleur(s)). Vous veillerez donc à séparer clairement le modèle de données (la liste des éléments ou les

éléments) des vues (l'affichage de la liste et des éléments et l'élément courant) et des contrôleurs (les widgets et/ou les délégués permettant de modifier le(s) modèle(s)). Chaque classe devra être documentée (avec Doxygen) et son auteur identifié.

1) Concevez les classes permettant de représenter votre modèle de données. Les classes directement en relation avec l'interface graphique pourront être implémentée sous la forme de QObjects afin de fournir des slots à l'interface et des signaux vers l'interface.
2) Concevez l'interface graphique en QT permettant de gérer les éléments de votre modèle.
   a) Définissez clairement chaque fonctionnalité (dans une action par exemple).
3) Concevez les classes nécessaires à l'importation et l'exportation des éléments de votre modèle.

Échéancier

- 25/02/2020 : Présentation du projet, choix d'un sujet et formation des binômes.
- 04/03/2020 : Validation du modèle sous forme d'un graphe de classe (modèle seul).
- 27/03/2020 : Dépôt de votre code sur le serveur de dépôt avec un rapport détaillant :
    - Le graphe de classes de votre modèle.
    - Les fonctionnalités de votre application et comment on peut les utiliser (menu, bouton, curseur, etc.).

Grille de notation

La grille de notation contiendra les éléments suivants :

- Classes du modèle
    - Hiérarchie d'héritage des différents types d'éléments
    - Composition des éléments
    - Signaux / Slots pour l'interaction entre le modèle et l'interface graphique
    - Type de modèle QT sous-jacent.
- Fonctionnalités
    - Affichage de la liste des éléments (avec evt un tri si l'application s'y prête) : partie gauche.
    - Affichage des détails d'un élément sélectionné : partie droite.
    - Edition d'un élément sélectionné : partie droite.
    - Création d'un nouvel élément.
    - Destruction du ou des éléments sélectionnés.
    - Recherche (filtrage) d'un ou plusieurs éléments particuliers dans la liste des éléments.
    - Enregistrement du modèle dans un fichier.
    - Lecture (ou ajout) d'un modèle depuis un fichier.
- Interface graphique
    - Layouts.
    - Sub-Components pour l'édition des détails d'un élément.
    - In-place Edit (→ évitez les boites de dialogue).
    - Structure dynamique (synchronisation du modèle avec l'interface graphique → signaux / slots).
    - Contrôle de saisie (en utilisant si besoin des QValidators).
    - Utilisation de la barre d'état.

5. Conseils

- Les classes directement en relation avec l'interface graphique pourront être implémentées sous la forme de QObjects afin de fournir des slots à l'interface et des signaux vers l'interface. La classe contenant les données devra respecter l'interface du modèle choisi.
- Évitez d'ouvrir une boite de dialogue à chaque fois que vous souhaitez modifier un champ, préférez l'édition directe du champ (grâce à l'éditeur d'un délégué par exemple).

- Lors de l'édition d'un champ, utilisez un QValidator (ou un de ses descendants) pour vérifier la validité de ce que vous tapez.
- Si vous développez vos propres widgets (pour la visualisation et l'édition d'un élément par exemple), n'oubliez pas de lui associer des propriétés (qui apparaîtront alors dans le designer) et des signaux/slots pour les connecter au(x) modèle(s).
- Utilisez la petite application qui vous est fournie (dans /pub/LOA/CookingClock.tgz) pour vous aider à :
    - Rendre votre application réellement multiplateforme.
    - Mettre en place une icône pour l'application (sur toute plateforme).
    - Charger et sauvegarder les préférences de l'application.
    - Traduire votre application dans différentes langues et changer la langue de votre application à la volée.
    - Mettre en place une boite de dialogue « à propos … » permettant de présenter l'application, ses auteurs et les crédits nécessaires si vous avez utilisé des ressources comme des icônes ou des librairies par exemple.
    - Avoir un exemple de code documenté avec Doxygen.
    - Avoir un exemple de fichier de configuration (.pro) relativement complet vous permettant (en plus de compiler votre application) de :
        - Générer une archive transportable de votre code (que vous pourrez me rendre à la fin du projet).
        - Générer la documentation de votre code.
        - Générer le listing de votre code.
        - Générer et compiler les fichiers de traduction multilingues.

Annexes

Documentation

- Documentation QT : http://doc.qt.io/qt-5/
    - Ensemble des tutoriaux QT : http://doc.qt.io/qt-5/qtexamplesandtutorials.html
    - Model/View Programming : http://doc.qt.io/qt-5/model-view-programming.html
        - Tutoriel Model / View : http://doc.qt.io/qt-5/modelview.html
        - Une version simplifiée mais en français a été réalisée par Thierry Vaira : http://tvaira.free.fr/dev/qt/coursQt-ModelView.pdf.
- Sites contenant des exemples et des tutoriaux QT
    - http://qt-apps.org/ (en anglais)
    - http://qt.developpez.com/ (en français)

Exemples de modèles de données

Gestionnaire de contacts

Un contact est un organisme ou une personne (qui peut-elle même faire partie d'un ou plusieurs organismes). Une personne est définie par un nom, éventuellement suivi par d'autres champs qui caractérisent cette personne (sexe, tel., adresse, email, … ou tout autre champ que vous jugerez pertinent). Un organisme est défini par son nom, sa raison sociale, ses coordonnées, éventuellement suivis par d'autres champs que vous jugerez pertinents. Certains champs sont constitués d'un seul élément, alors que d'autres sont constitués de plusieurs éléments (éventuellement de types différents).

Liste (non exhaustive) des champs associés à une personne ou une organisation :

Personne

Champ	Type	arité	Sous-champ	Type	arité	Exemple
nom	stucturé	1	Nom	texte	1	Roussel
			Prénom	texte	1..*	David
			Surnom	texte	0..1	
			Préfixe	texte	0..*	Dr.
			formattage	structure	1	Dr. David Roussel
sexe	enum	0..1		enum : {homme, femme}		homme
tel	structuré	0..*	type	enum : {work, home, ...}	1	work
			type tel	enum : {fixe, cellulaire, ...}	1	fixe
			type données	enum : {voice, data, ...}	1	voice
			n°	texte ou structure	1	01 69 36 74 62
adresse	structuré	0..*	type	enum : {work, home, ...}	1	work
			rue	texte	1..*	1 place de la Résistance
			zipcode	texte ou structure	1	91025
			localité	texte	1	Evry cedex
			Région	texte	0..1	
			pays	texte	1	France
			geo	latitude + longitude	0..1	48.626512, 2.432210
email	structuré	0..*	nom	texte	1	david.roussel
			domain	texte	1	ensiie
			extension	texte	1	fr
site	structuré	0..*	type	enum : {web, facebook, go	1	web
			url	url	1	http://www.ensiie.fr
photo	image	0..1	type	enum : {JPEG, GIF, ...}	1	
			url	url	1	
organisation	card	0..*				card de l'ENSIIE
type	texte ou enum	0..*				collègue, ami
date MAJ	timestamp	1				
note	texte	0..*				

Organisation

Champ	Type	arité	Sous-champ	Type	arité
nom	stucturé	1	Nom	texte	1
			raison sociale	...	0..1
			...	...	...
tel	structuré	0..*	type tel	enum : {fixe, cellulaire, ...}	1
			type données	enum : {voice, data, ...}	1
			n°	texte ou structure	1
adresse	structuré	0..*	intitulé	texte	0..1
			rue	texte	1..*
			zipcode	texte ou structure	1
			localité	texte	1
			Région	texte	0..1
			pays	texte	1
			geo	latitude + longitude	0..1
email	structuré	0..*	nom	texte	1
			domain	texte	1
			extension	texte	1
site	structuré	0..*	type	enum : {web, facebook, google+, ...}	1
			url	url	1
logo	image	0..1	type	enum : {JPEG, GIF, ...}	1
			url	url	1
membre	structuré	0..*	card	card d'une personne	1
			fonction	texte	1
date MAJ	timestamp	1			
note	texte	0..*			

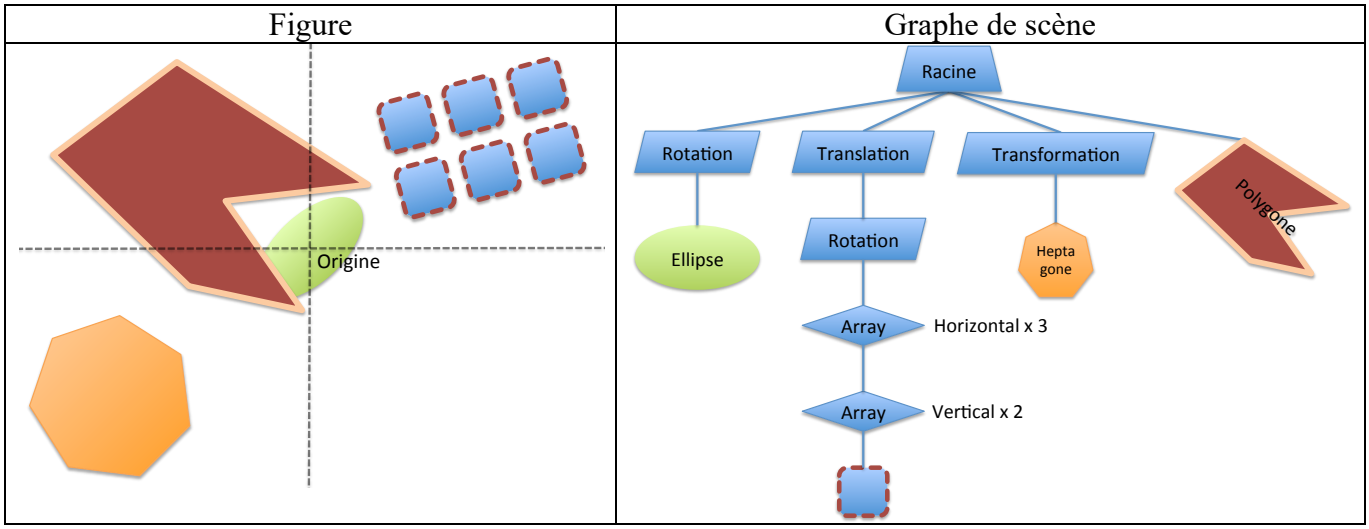
On pourra avantageusement utiliser un QDataWidgetMapper évoqué dans le tutoriel Model / View de QT pour éditer un contact dans la partie droite de l'UI.

Graphe de scène 2D pour un éditeur de figures géométriques

Un graphe de scène (2D ou 3D) est composé de nœuds reliés entre eux par des relations de parenté. On trouve dans un graphe de scène deux types de nœuds :

- Les nœuds terminaux qui représentent les figures à dessiner :
  - Cercle, ellipse, rectangle, trapèze, polygone irrégulier ou polygones réguliers (n-gones).
  - Chaque figure géométrique est caractérisée par son type, mais aussi par sa couleur de remplissage (couleur unie ou gradient de couleurs) ainsi que par son contour (couleur du trait, type de trait (continu ou pointillé) et épaisseur du trait). Voir les exemples suivants :
    - http://doc.qt.io/qt-5/qtwidgets-painting-basicdrawing-example.html
    - http://doc.qt.io/qt-5/paintsystem-drawing.html
- Les nœuds non terminaux qui représentent les nœuds de groupement. Ce sont donc des nœuds composites qui contiennent d'autres nœuds dont :
  - Les nœuds de groupement simples qui ne font que grouper leurs enfants.
  - Les nœuds de « transformation » (translation, rotation, échelle) : ces nœuds contiennent une transformation géométrique qui s'appliquera sur tous ses enfants.
  - Les nœuds « array » qui permettent de dupliquer le contenu plusieurs fois suivant un axe particulier.
  - Les nœuds « select » qui permettent de n'afficher qu'un seul de leurs enfants.

Voici un exemple de dessin utilisant un graphe de scène :



On remarquera que le polygone irrégulier recouvre partiellement l'ellipse ce qui suppose que le graphe de scène est parcouru de gauche à droite lors du dessin.

Éditeur d'images (ou comment améliorer un modèle trop simple)

Si le modèle est constitué d'une collection d'images cela ne constitue pas un modèle suffisamment intéressant en dehors de l'affichage de chaque image et/ou des caractéristiques de celles-ci (type, taille, métadonnées, etc.). En revanche si on lui ajoute la composition d'images dans des opérateurs binaires (multiplication, min, max, …) ou des opérateurs unaires (rotation des couleurs, ajustement du contraste, …). A ce moment là seulement, le modèle devient suffisamment intéressant pour être implémenté dans le cadre de ce projet.

Voir l'exemple de l'éditeur d'images QT :
http://doc.qt.io/qt-5/qtwidgets-widgets-imageviewer-example.html.